

COMPUTER SCIENCE

with

[A Textbook for Class XI]

I
S
C

JAVA

SUMITA ARORA

DHANPAT RAI & CO. (Pvt.) Ltd.
EDUCATIONAL & TECHNICAL PUBLISHERS

**Published by : GAGAN KAPUR
Dhanpat Rai & Co. (P) Ltd., Delhi**

*Regd. Office : 4576/15, Agarwal Road
Darya Ganj, New Delhi-110002
Phone : 2324 7736, 37, 38, dhanpatrai@gmail.com*

© Sumita Arora

All Trademarks Acknowledged

Disclaimer

Every effort has been made to avoid errors or omissions in this publication. In spite of this, some errors might have crept in. Any mistake, error or discrepancy noted may be brought to our notice which shall be taken care of in the next edition. It is notified that neither the publisher nor the author or seller will be responsible for any damage or loss of action to any one, of any kind, in any manner, therefrom.

EDITIONS : 2002, 2003, 2007, 2008, 2015, 2016

REPRINT : 2010, 2011, 2012, 2013, 2014, 2017, 2019, 2020, 2021, 2022, 2023

SEVENTH EDITION : 2018

REPRINT : 2024

PRICE : ₹ 595/-

*Typesetting by : North Delhi Computer Convention, Delhi-110009,
ndcc.in@gmail.com*

Printed at : Natraj Offset, Delhi

Contents



Chapter 1

DATA REPRESENTATION

1 - 61

1.1	Introduction	1
1.2	Numeral Systems	1
1.2.1	Categories of Number Systems	2
1.3	Digital Number Systems	5
1.3.1	Decimal Number System	5
1.3.2	Binary Number System	5
1.3.3	Octal Number System	7
1.3.4	Hexadecimal Number System	7
1.4	Number Conversions	8
1.4.1	Decimal-to-Binary Conversion	9
1.4.2	Binary-to-Decimal Conversion	13
1.4.3	Decimal-to-Octal Conversion	15
1.4.4	Octal-to-Decimal Conversion	15
1.4.5	Octal-to-Binary Conversion	15
1.4.6	Binary-to-Octal Conversion	16
1.4.7	Decimal-to Hex Conversion	17
1.4.8	Hex-to-Decimal Conversion	17
1.4.9	Binary-to-Hex Conversion	18
1.4.10	Hex to Binary Conversion	18
1.5	Binary Representation of Numbers	20
1.5.1	Binary Representation of Integers	21
1.6	Floating-Point Representation (For Real Numbers)	27
1.6.1	Fixed-Point Representation	27
1.6.2	Scientific Notation (Exponential Notation)	28
1.6.3	Normalized Scientific Notation	28
1.6.4	Mantissa-Exponent Notation	29
1.6.5	Single and Double Precision Representation Schemes	32
1.7	Binary Arithmetic	33
1.7.1	Binary Addition	33
1.7.2	Binary Subtraction	35
1.7.3	Binary Addition and Subtraction using One's Complement Notation	37
1.7.4	Binary Addition and Subtraction using 2's complement Notation	39
1.7.5	Binary Multiplication	41
1.7.6	Binary Division	42

1.8 Adding and Subtracting Octal Numbers	42
1.8.1 Octal Addition	43
1.8.2 Octal Subtraction	44
1.9 Adding and Subtracting Hexadecimal Numbers	45
1.9.1 Adding two Hexadecimal Numbers	45
1.9.2 Hexadecimal Subtraction	47
1.10 Representing Characters in Memory	48
1.11 ASCII	50
1.12 Unicode	52

Chapter 2 PROPOSITIONAL LOGIC & HARDWARE 63 – 96

2.1 Introduction	63
2.2 Propositional Logic	63
2.2.1 Terms and Symbols	64
2.2.2 Truth values and Wff	65
2.2.3 Some Equivalence Propositional Laws	68
2.2.4 Drawing Conclusions – Syllogism	71
2.3 Basic Logic Gates	74
2.3.1 Inverter (NOT Gate)	75
2.3.2 OR Gate	75
2.3.3 AND gate	76
2.4 More About Logic Gates	76
2.4.1 NOR Gate	76
2.4.2 NAND Gate	77
2.4.3 XOR Gate (Exclusive OR Gate)	78
2.4.4 XNOR Gate (Exclusive NOR gate)	79
2.4.5 NAND to NAND and NOR to NOR Design	80
2.5 Applications of Logic Gates	84
2.5.1 Adders	84

Chapter 3 GENERAL OOP CONCEPTS 97 – 108

3.1 Introduction	97
3.2 Evolution of Software	98
3.2.1 Programming Paradigms	98
3.3 Basic Concepts of OOP	101
3.3.1 Data Abstraction	101
3.3.2 Encapsulation	101
3.3.3 Modularity	102
3.3.4 Inheritance	103
3.3.5 Polymorphism	104

4.1	Introduction	109
4.2	About Java	109
4.2.1	<i>History of Java</i>	110
4.2.2	<i>Byte Code</i>	110
4.2.3	<i>Java Virtual Machine (JVM)</i>	112
4.2.4	<i>Characteristics of Java</i>	112
4.3	Simple Java Program	113
4.3.1	<i>Types of Java Programs</i>	114
4.4	Creating and Running a Java Program	115
4.4.1	<i>General Method of Creating and Running Java Programs</i>	115
4.4.2	<i>Creating and Running Java Program Using JCreator LE</i>	117

5.1	Introduction	123
5.2	Java Character Set	123
5.3	Tokens	124
5.3.1	<i>Keywords</i>	124
5.3.2	<i>Identifiers</i>	125
5.3.3	<i>Literals</i>	126
5.3.4	<i>Separators</i>	129
5.3.5	<i>Operators</i>	129
5.4	Concept of Data Types	129
5.4.1	<i>Primitive Datatypes</i>	130
5.4.2	<i>Reference Datatypes</i>	133
5.5	Variables	135
5.5.1	<i>Declaration of a Variable</i>	135
5.5.2	<i>Initialization of Variables</i>	136
5.6	Constants	139
5.7	Operators in Java	141
5.7.1	<i>Arithmetic Operators</i>	141
5.7.2	<i>Increment/Decrement Operators (++, --)</i>	142
5.7.3	<i>Relational Operators</i>	144
5.7.4	<i>Logical Operators</i>	145
5.7.5	<i>Shift Operators</i>	147
5.7.6	<i>Bitwise Operators</i>	149
5.7.7	<i>Assignment Operators</i>	151
5.7.8	<i>Other Operators</i>	153
5.7.9	<i>Operator Precedence</i>	155
5.8	Expressions	157
5.8.1	<i>Arithmetic Expressions</i>	158

5.9 Java Statements	163
5.10 Significance of Classes	165
5.11 Objects as Instances of Class	171

Chapter 6 FLOW OF CONTROL 185 - 242

6.1 Introduction	185
6.2 Programming Constructs	185
6.3 Selection Statements	187
6.3.1 <i>The if Statement of Java</i> 187	
6.3.2 <i>The switch Statement</i> 199	
6.4 Iteration Statements	208
6.5 Elements that Control a Loop (Parts of a Loop)	208
6.6 The for Loop – Fixed Number of Iterations	209
6.6.1 <i>The for Loop Variations</i> 212	
6.7 The while Loop	214
6.7.1 <i>Variations in a while Loop</i> 215	
6.8 The do-while Loop	216
6.9 Nested Loops	217
6.10 Comparison of Loops	218
6.11 Jump Statements	218
6.11.1 <i>The break Statement</i> 218	
6.11.2 <i>The continue Statement</i> 220	
6.11.3 <i>Labels and Branching Statements</i> 221	

Chapter 7 CLASSES IN JAVA 243 - 264

7.1 Introduction	243
7.2 Class as Composite Type	244
7.2.1 <i>Class as User-defined Datatype</i> 245	
7.3 Creating and Using Objects	246
7.3.1 <i>Using Objects</i> 248	
7.3.2 <i>Controlling Access to Members of a Class – Access Specifiers</i> 250	
7.4 Encapsulation	254
7.5 Visibility Modifiers	255
7.6 Scope and Visibility Rules	255

Chapter 8 FUNCTIONS (METHODS) 265 - 312

8.1 Introduction	265
8.2 Why Functions ?	266
8.3 Function/Method Definition	267
8.3.1 <i>Function Prototype and Signature</i> 269	

8.4 Accessing a Function	271
8.4.1 Actual and Formal Parameters	273
8.4.2 Arguments to Functions	273
8.5 Pass By Value (Call By Value)	274
8.6 Call By Reference	275
8.7 Returning from a Function	282
8.7.1 The return statement	283
8.7.2 Returning Values	283
8.8 Pure and Impure Functions	284
8.9 Function Overloading	285
8.9.1 Need For Function Overloading	285
8.9.2 Declaration and Definition	287
8.10 Calling Overloaded Functions	288
8.11 Constructors	289
8.11.1 Need for Constructors	290
8.11.2 Declaration and Definition	290
8.12 Types of Constructors	291
8.12.1 Non-Parameterized Constructors	292
8.12.2 Parameterized Constructors	292
8.13 The this Keyword	294
8.14 Temporary Instances	296
8.15 Constructor Overloading	297
8.16 Recursive Functions	298
8.17 Recursion in Java	299

 Chapter 9	PROGRAM ERROR TYPES AND BASIC EXCEPTION HANDLING	313 – 338
9.1 Introduction		313
9.2 Type of Program Errors		314
9.2.1 Compile-Time Errors	314	
9.2.2 Run-Time Errors	316	
9.2.3 Logical Errors	317	
9.3 Exception and Exception Handling		319
9.3.1 What happens when an Exception occurs ?	321	
9.3.2 Concept of Exception Handling	322	
9.3.3 Exception Handling in Java	323	
9.3.4 The Exception Class	327	
9.3.5 Common Exceptions	328	
9.4 Exception Hierarchy		329
9.5 Forcing an Exception		331
9.6 Benefits of Exception Handling		332

Chapter 10 USING LIBRARY CLASSES AND PACKAGES 339 – 374

10.1 Introduction	339
10.2 Simple Input/Output	340
10.3 Exception Handling	348
10.3.1 Concept of Exception Handling	348
10.3.2 Exception Handling in Java	349
10.4 Wrapper Classes	352
10.5 Working with Strings	354
10.5.1 Creating Strings	354
10.5.2 Creating StringBuffers	354
10.5.3 Accessor Methods	355
10.6 Packages in Java	363
10.6.1 Importing Packages and their Classes	364
10.6.2 Using Dates and Times (Date and Calendar Objects)	364
10.6.3 Packages in Java	366
10.6.4 User Defined Packages	367

Chapter 11 ARRAYS 375 – 416

11.1 Introduction	375
11.2 Need for Arrays	376
11.3 Types of Arrays	376
11.3.1 Single Dimensional Arrays	376
11.3.2 Some Facts about Arrays	381
11.3.3 Two-Dimensional Arrays	382
11.4 Searching in 1-D Arrays	387
11.5 Sorting	392
11.6 Arrays vs. Objects	397
11.7 Advantages and Disadvantages of Arrays	398
11.8 Solution of Simultaneous Linear Equations	399
11.8.1 Gauss Elimination Method	399
11.8.2 Generalised Solution by Gauss Elimination Method and Corresponding Computer Program	403

Chapter 12 OPERATIONS ON FILES 417 – 451

12.1 Introduction	417
12.2 Files	417
12.3 Java Streams	418
12.4 Operations on Files	419
12.4.1 Buffering	419

12.4.2	<i>Output to Text Files</i>	420
12.4.3	<i>Input from Text Files</i>	423
12.4.4	<i>Output to Binary Files</i>	424
12.4.5	<i>Input from Binary Files</i>	426
12.5	String Tokenizer	428
12.5.1	<i>Processing a Number Sequence with StringTokenizer</i>	430
12.5.2	<i>String Tokenizing and File Handling</i>	431
12.6	Stream Tokenizer	432
12.7	Obtaining Input using Scanner Class	435
12.8	Printing in Java	439
12.8.1	<i>Java Printing API</i>	439

 Chapter 13 **TRENDS IN COMPUTING AND ETHICAL ISSUES** **453 – 475**

13.1	Introduction	453
13.2	Trends in Computing	454
13.2.1	<i>Artificial Intelligence</i>	454
13.2.2	<i>Internet of Things (IoT)</i>	454
13.2.3	<i>Virtual Reality (VR)</i>	456
13.2.4	<i>Augmented Reality (AR)</i>	457
13.3	Cyber Security	458
13.3.1	<i>Threats to Computer Security</i>	458
13.3.2	<i>Cyber Security Measures</i>	460
13.3.3	<i>Online Privacy</i>	462
13.4	Major Ethical Issues	464
13.4.1	<i>Individual's Right to Privacy</i>	464
13.4.2	<i>Intellectual Property Rights</i>	465
13.4.3	<i>Accuracy of Information</i>	465
13.5	Open Source and Its Philosophy	465
13.5.1	<i>Terminology</i>	466
13.5.2	<i>Philosophy of Open Source</i>	466
13.5.3	<i>Definitions</i>	467
13.5.4	<i>Licenses and Domains of Open Source Technology</i>	469
13.6	Netiquette	471
13.7	Email Etiquette	471
Appendices		476-480

Data Representation

1.1 Introduction

Digital techniques have found their way into innumerable areas of technology, but the area of automatic digital computers is by far the most notable and most extensive. As you know, a computer is a system of hardware that performs arithmetic operations, manipulates data, and makes decisions.

In science, technology, business, and, in fact, most other fields of endeavour, we are constantly dealing with quantities; so are computers. Quantities are measured, monitored, recorded, manipulated arithmetically, observed, or in some other way utilized in most physical systems. In digital systems like computers, the quantities are represented by symbols called digits. Many number systems are in use in digital technology that represent the digits in various forms. The most common are the decimal, binary, octal, and hexadecimal systems. This chapter discusses these number systems and the physical representation of digits in computers.

1.2 Numeral Systems

Numeral systems or *Numbering systems* or simply *Number Systems* are a way of representing numbers by a finite set of distinct graphic symbols or signs. Same number can be represented differently in different numeral systems. For instance, number 42 of decimal system can also be represented as 52 in octal number system or as 2A in hexadecimal number system.

In This Chapter

- 1.1 Introduction
- 1.2 Numeral Systems
- 1.3 Digital Number Systems
- 1.4 Number Conversions
- 1.5 Binary Representation of Numbers
- 1.6 Floating – Point Representation
- 1.7 Binary Arithmetic
- 1.8 Adding and Subtracting Octal Numbers
- 1.9 Adding and Subtracting Hexadecimal Numbers
- 1.10 Adding and Subtracting Floating-Point Numbers
- 1.11 Limitations of Finite Representation
- 1.12 Representing Characters in Memory
- 1.13 ISCII
- 1.14 Unicode

1.2.1 Categories of Number Systems

Several number systems have been used in past. Currently also multiple number systems are in use. These number systems can be broadly divided into *two* categories :

- ❖ Non-positional number systems
- ❖ Positional number systems

Let us briefly discuss about their difference and then we shall continue with positional number systems as per the syllabus.

1.2.1A Non-Positional Numeral/Number Systems

In early days, people counted using fingers, stones, pebbles or sticks etc. Non-positional number systems are based on this method of counting and additive approach. In non-positional number system, a symbol carries the same value always irrespective of its position.

Two such number systems are : **Roman Number System** and **Greek Number System**.

Roman Numeral System

One popular non-positional number system is the **Roman numeral system**, which uses sequences of the following symbols to represent the numbers :

<i>Roman Digit</i>	<i>Decimal Value</i>	<i>Roman Digit</i>	<i>Decimal Value</i>
I	1	C	100
V	5	D	500
X	10	M	1000
L	50		

As this is non-positional number system, the position of the digits has no significance for the value of the number depicted in this number system.

For determining the value of a Roman number, the following rules are applied :

- If two consecutively represented Roman digits are in such order that the value of the first one is bigger or equal to the value of the second one, their values are added.

For example :

$$\text{III} = 3$$

[values of three I's added]

$$\text{MD} = 2500$$

[values of M, M and D added]

- If two consecutively represented roman digits are in increasing order of their values, they are subtracted.

For example :

$$\text{IX} = 9$$

[value of I subtracted from value of X]

$$\text{MXL} = 1040$$

[value of X subtracted from value of L, giving 40 which is added to M]

$$\text{CM} = 900$$

[value of C subtracted from value of M]

Greek Numeral System

Another non-positional number system is Greek numeral system. In this number system, grouping of fives is done. It uses the following digits :

<i>Greek Digit</i>	<i>Decimal Value</i>	<i>Greek Digit</i>	<i>Decimal Value</i>
I	1	H	100
Γ	5	X	1,000
Δ	10	M	10,000

As we can see in the table, one is represented with a vertical line, five with the letter Γ, and the powers of 10 with the first letter of the corresponding Greek word.

Here are some examples of numbers in this system :

$$\Gamma\Delta = 50 = 5 \times 10$$

$$\Gamma H = 500 = 5 \times 100$$

$$\Gamma X = 5000 = 5 \times 1,000$$

$$\Gamma M = 50,000 = 5 \times 10,000$$

In a non-positional number system, the placement of a numeral digit does not change its value.

1.2.1B Positional Numeral/Number Systems

Positional number systems (also called **place-value numeral systems**) are systems in which the placement of a digit in connection with its intrinsic value determines its actual weight or value in the number being presented.

For instance,

$$\text{In } 135, \text{ 5 has value } 5 \times 1 = 5$$

$$153, \text{ 5 has value } 5 \times 10 = 50$$

$$531, \text{ 5 has value } 5 \times 100 = 500$$

As you can see in above examples that digit '5' is in all three numbers but each digit '5' carries different weight (actual value) depending upon its position in the number. Before we proceed, let us know about an important term related to positional number system – Base or Radix.

Base or Radix

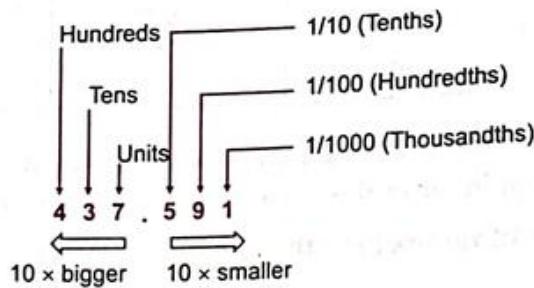
This is an important term associated with positional number systems. A Base or Radix tells how many distinct graphic symbols are used to represent number in a number system. For example, in decimal number system, the base is 10 as it uses ten (10) different symbols {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} to represent numbers. Whatever mathematics you have studied so far, has been based on Decimal number system. So you can safely say that you have been writing and using numbers based on Decimal number system.

This *base* or *radix* plays an important role in determining positional value of a digit.

Number Representation in Positional Numeral Systems

In Positional number systems¹, a number is represented in a way that depicts the positional value of each digit, using base or radix of the number system. For example, consider decimal number system, you are familiar with Decimal number system, which you have been working with since you started schooling.

The decimal number system represents numbers where each digit has a place value, e.g.,



If you just consider the non-fractional part 437 of above number, you can represent it as

$$4 \times 10^2 + 3 \times 10^1 + 7 \times 10^0$$

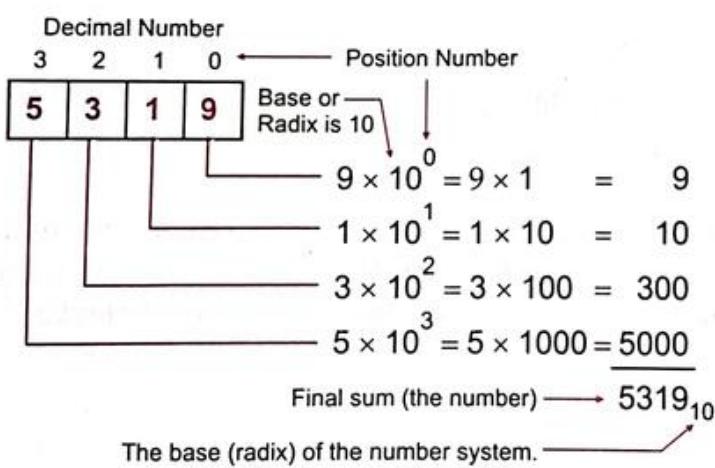
As you can make out the positional value of 4 becomes $4 \times 100 = 400$; positional value of 3 becomes $3 \times 10 = 30$ and positional value of 1 is $1 \times 1 = 1$.

Similarly, you can represent its fractional part .591 as : $5 \times 10^{-1} + 9 \times 10^{-2} + 1 \times 10^{-3}$

So the total number is represented as : $4 \times 10^2 + 3 \times 10^1 + 7 \times 10^0 + 5 \times 10^{-1} + 9 \times 10^{-2} + 1 \times 10^{-3}$

You must have noticed that each digit's positional value depends upon the position of a digit multiplied with a power of base (radix) of the number system (base in this case is 10).

Following figure also illustrates the same :



Note In a positional number system, the position a digit occupies multiplied by a power of the radix of the number system determines its actual value.

Figure 1.1 Decimal Number System – a Positional Number System.

In the coming lines, we shall be covering four positional number systems — decimal, binary, octal and hexadecimal which are commonly used in digital calculations.

1. In fixed-radix positional number systems only ; there can be mixed-radix positional number systems also. But here we shall stick to discussion of fixed-radix number systems as discussion of other types of positional number systems is beyond the scope of this book.

1.3 Digital Number Systems

In digital representation, various number systems are used. The most common number systems used are *decimal*, *binary*, *octal*, and *hexadecimal* systems. Let us discuss these number systems briefly.

1.3.1 Decimal Number System

The *decimal system* is composed of 10 numerals or symbols (*Deca* means 10, that is why this system is called *decimal system*). These 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ; using these symbols as *digits* of a number, we can express any quantity. The decimal system, also called the *base-10* system because it has 10 digits, has evolved naturally as a result of the fact that man has 10 fingers.

The decimal system is a *positional-value* system in which the value of a digit depends on its position. For example, consider the decimal number 729. We know that the digit 7 actually represents 7 *hundreds* the 2 represents 2 *tens* and the 9 represents 9 *units*. In essence, the 7 carries the most weight of three digits; it is referred to as the *most significant digit* (MSD). The 9 carries the least weight and is called the *least significant digit* (LSD).

Consider another example, 25.12. This number is actually equal to (2 *tens* plus 5 *units* plus 1 *tenths* plus 2 *hundredths*) i.e., $2 \times 10 + 5 \times 1 + 1 \times \frac{1}{10} + 2 \times \frac{1}{100}$. The decimal point is used to

separate the integer and fractional parts of the number.

More rigorously, the various positions relative to the decimal point carry weights that can be expressed as powers of 10. This is illustrated in Fig. 1.2 where the number 2512.1971 is represented. The decimal point separates the positive powers of 10 from the negative powers. The number 2512.1971 is thus equal to

$$2 \times 10^3 + 5 \times 10^2 + 1 \times 10^1 + 2 \times 10^0 + 1 \times 10^{-1} + 9 \times 10^{-2} + 7 \times 10^{-3} + 1 \times 10^{-4}$$

In general, any number is simply the sum of the products of each digit value and its positional value

10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}
↓	↓	↓	↓	↓	↓	↓	↓
2	5	1	2	1	9	7	1
↑			↑	decimal point		↑	
MSD						LSD	

Figure 1.2 Positional values in decimal numbers.

The sequence of decimal numbers goes as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21.... See after 9, each successive number is a combination of two (or more) (unique) symbols of this system.

1.3.2 Binary Number System

Unfortunately, the decimal number system does not lend itself to convenient implementation in digital systems. For example, it is very difficult to design electronic equipment so that

it can work with 10 different voltage levels (each one representing one decimal character, 0 through 9). On the other hand, it is very easy to design simple, accurate electronic circuits that operate with only two voltage levels. For this reason, almost every digital system uses the binary number system (base 2) as the basic number system of its operations, although other systems are often used in conjunction with binary.

In the *binary system* there are only two symbols or possible digit values, 0 and 1. Even so, this base-2 system can be used to represent any quantity that can be represented in decimal or other number systems.

The binary system is also a positional-value system, wherein each binary digit has its own value or weight expressed as a power of 2. This is illustrated in Fig. 1.3.

Here, places to the left of the binary point (counterpart of the decimal point) are positive powers of 2 and places to the right are negative powers of 2. The number 1010.0101 is shown represented in the figure

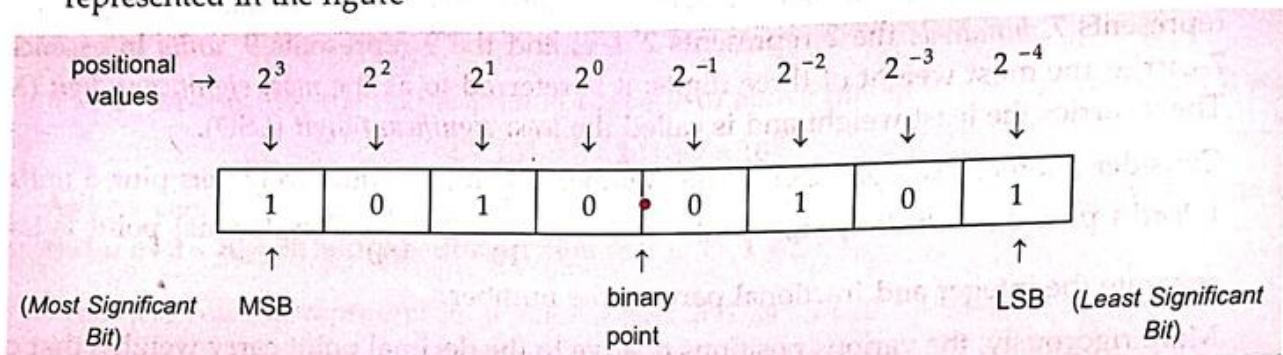


Figure 1.3 Positional values in binary numbers.

To find the decimal equivalent of above shown binary number, we simply take the sum of the products of each digit value (0 or 1) and its positional value :

$$\begin{aligned}
 1010.0101_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4}) \\
 &= 8 + 0 + 2 + 0 + 0 + 0.25 + 0 + 0.0625 \\
 &= 10.3125_{10}
 \end{aligned}$$

Notice in the preceding operation that subscripts (2 and 10) were used to indicate the base in which the particular number is expressed. This convention is used to avoid confusion whenever more than one number system is being employed.

In the binary system, the term *Binary digit* is often abbreviated to the term *bit*, which we'll use henceforth. As you see in Fig. 1.3, there are 4 bits to the left of the binary point, representing the integer part of the number, and 4 bits to the right of the binary point, representing the fractional part. The leftmost bit carries the largest weight and hence, is called the most significant bit (MSB). The rightmost bit carries the smallest weight, and hence called least significant bit (LSB).

The sequence of binary numbers goes as 00, 01, 10, 11, 100, 101, 110, 111, 1000, ----- . The binary counting sequence has an important characteristic. The units bit (LSB) changes either from 0 to 1 or 1 to 0 with each count. The second bit (two's (2^1) position) stays at 0 for two counts, then at 1 for two counts, then at 0 for two counts, and so on. The third bit (four's (2^2) position) stays at 0 for four counts, then at 1 for four counts, and so on. The fourth bit

(eight's (2^3) position) stays at 0 for eight counts, then at 1 for eight counts. If we wanted to count further we would add more places, and this pattern would continue with 0s and 1s alternating in groups of 2^{N-1} .

1.3.3 Octal Number System

The octal number system is very important in digital computer work. The octal number system has a base of *eight*, meaning that it has eight unique symbols : 0, 1, 2, 3, 4, 5, 6, and 7. Thus, each digit of an octal number can have any value from 0 to 7.

The octal system is also a positional value system, wherein each octal digit has its own value or weight expressed as a power of 8 (See Fig. 1.4). The places to the left of the octal point (counter-part of decimal point and binary point) are positive powers of 8 and places to the right are negative powers of 8. The number 3721.2406 is shown represented in the figure.

positional values (weights)	8^3	8^2	8^1	8^0	8^{-1}	8^{-2}	8^{-3}	8^{-4}
→ ↓	3	7	2	1	• 2	4	0	6

↑
Octal point

Figure 1.4 Positional values in Octal numbers.

To find the decimal equivalent of above shown octal number, simply take the sum of products of each digit value and its positional value :

$$\begin{aligned}
 3721.2406_8 &= (3 \times 8^3) + (7 \times 8^2) + (2 \times 8^1) + (1 \times 8^0) + (2 \times 8^{-1}) + (4 \times 8^{-2}) + (0 \times 8^{-3}) + (6 \times 8^{-4}) \\
 &= 3 \times 512 + 7 \times 64 + 2 \times 8 + 1 \times 1 + 2 \times 0.125 + 4 \times 0.015625 + 0 + 6 \times 0.000244 \\
 &= 1536 + 448 + 16 + 1 + 0.25 + 0.0625 + 0 + 0.001464 \\
 &= 2001.313964_{10}
 \end{aligned}$$

The sequence of octal numbers goes as 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22..... See each successive number after 7 is a combination of 2 or more unique symbols of octal system.

1.3.4 Hexadecimal Number System

The hexadecimal system uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F as the 16 digit symbols.

Just like above discussed systems, hexadecimal system is also a positional-value system, wherein each hexadecimal digit has its own value or weight expressed as a power of 16. (See Fig. 1.5). The digit positions in a hexadecimal number have weights as follows :

Weights →	16^3	16^2	16^1	16^0	$• 16^{-1}$	16^{-2}	16^{-3}	16^{-4}
-----------	--------	--------	--------	--------	-------------	-----------	-----------	-----------

Hexadecimal point

Figure 1.5 Positional values in hexadecimal number.

Following table 1.1 shows the relationships between hexadecimal, octal, decimal and binary.

Table 1.1 Relationship between various number systems

Hexadecimal	Octal	Decimal	Binary
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	10	8	1000
9	11	9	1001
A	12	10	1010
B	13	11	1011
C	14	12	1100
D	15	13	1101
E	16	14	1110
F	17	15	1111

Note that each hexadecimal digit represents a group of four binary digits. It is important to remember that *hex* (abbreviation for hexadecimal) digits A through F are equivalent to the decimal values 10 through 15.

1.4 Number Conversions

The binary number system is the most important one in digital systems as it is very easy to implement in circuitry. The decimal system is important because it is universally used to represent quantities outside a digital system.

In addition to binary and decimal, octal and hexadecimal number systems find widespread application in digital systems. These number systems (octal and hexadecimal) provide an efficient means for representing large binary numbers. As we shall see, both these number systems have the advantage that they can be easily converted to and from binary.

In a digital system, three or four of these number systems may be in use at the same time, so that an understanding of the system operation requires the ability to convert from one number system to another. This section discusses how to perform these conversions. So, let us discuss them one by one.

1.4.1 Decimal-to-Binary Conversion

There are *two* procedures for converting (integers) from decimal to binary.

The first method requires a table of powers of 2 (Table 1.2). Because of this restriction, it is more useful for small numbers where these powers have been memorized. Starting with the decimal number to be evaluated, obtain the largest power of 2 from the table without exceeding the original number. Record this. Then subtract the table obtained number from the original number. Repeat the process for the remainder, and continue until the remainder is zero. Finally, add the binary numbers obtained from the table. The result is the answer.

Table 1.2 Powers of 2

2^n	n	2^{-n}											
1	0	1.0											
2	1	0.5											
4	2	0.25											
8	3	0.125											
16	4	0.062	5										
32	5	0.031	25										
64	6	0.015	625										
128	7	0.007	812	5									
256	8	0.003	906	25									
512	9	0.001	953	125									
1024	10	0.000	976	562	5								
2048	11	0.000	488	281	25								
4096	12	0.000	244	140	625								
8192	13	0.000	122	070	312	5							
16384	14	0.000	061	035	156	25							
32768	15	0.000	030	517	578	125							
65536	16	0.000	015	258	789	062	5						
131072	17	0.000	007	629	394	531	25						
262144	18	0.000	003	814	697	265	625						
524288	19	0.000	001	907	348	632	812	5					
1048576	20	0.000	000	953	674	316	406	25					
2097152	21	0.000	000	476	837	158	203	125					
4194304	22	0.000	000	238	418	579	101	562	5				
8388608	23	0.000	000	119	209	289	550	781	25				
16777216	24	0.000	000	059	604	644	775	390	625				
33554432	25	0.000	000	029	802	322	387	695	312	5			
67108864	26	0.000	000	014	901	161	193	847	656	25			
134217728	27	0.000	000	007	450	580	596	923	828	125			
268435456	28	0.000	000	003	725	290	298	461	914	062	5		
536870912	29	0.000	000	001	862	645	149	230	957	031	25		
1073741824	30	0.000	000	000	931	322	574	615	478	515	625		
2147483648	31	0.000	000	000	465	661	287	307	739	257	812	5	
4294967296	32	0.000	000	000	232	830	643	653	869	628	906	25	
8589934592	33	0.000	000	000	116	415	321	826	934	814	453	125	
17179869184	34	0.000	000	000	058	207	660	913	487	407	226	562	5
34359738368	35	0.000	000	000	029	103	830	456	733	703	613	281	25
68719476736	36	0.000	000	014	551	915	228	366	851	806	640	625	

10

Example 1.1. Convert 43_{10} to binary.

Solution. From Table 1.2, 32 is the largest number without exceeding 43.

$$32 = 100000$$

$(2^5 = 32$ i.e., put 1 at $(5+1)^{th}$ position and 0_s at all other positions. Thus $32_{10} = 100000_2$)

$$43 - 32 = 11$$

From the table, 8 (2^3) is the largest number without exceeding 11.

$$8 = 1000$$

$(2^3 = 8$ i.e., put 1 at $(3+1)^{th}$ position and 0_s at all other positions. Thus $8_{10} = 1000_2$)

$$11 - 8 = 3$$

From the table, 2 is the largest number without exceeding 3.

$$2 = 10$$

$(2^1 = 2$, thus $2_{10} = 10_2$)

$$3 - 2 = 1$$

1 is the largest number without exceeding 1

$$1 = 1$$

Add all the binary numbers obtained i.e.,

$$\begin{array}{r} 100000 \\ 1000 \\ 10 \\ 1 \\ \hline \underline{101011} \end{array}$$

Thus $43_{10} = 101011_2$

Example 1.2. Convert 200_{10} to binary.

Solution. First largest number is 128.

Longest number

$$128 = 10000000$$

$$200 - 128 = 72 \quad 64 = 1000000$$

$$72 - 64 = 8 \quad 8 = 1000$$

$$\underline{200 = 11001000}$$

Therefore, $200_{10} = 11001000_2$

The second method of converting decimal to binary is *repeated-division* method. In this method, the number is successively divided by 2 and its remainders recorded. The final binary result is obtained by assembling all the remainders, with the last remainder being the most significant bit (MSB).

Example 1.3. Convert 43_{10} to binary using repeated division method.

Solution.

Repeated Division		Remainders	
2	43		
2	21	1	
2	10	1	
2	5	0	
2	2	1	
2	1	0	
	0	1	
			MSB

Write in this order

Reading the remainders from the bottom to the top,

$$43_{10} = 101011_2 \quad (\text{compare with result of example 1.1})$$

Example 1.4. Convert 200_{10} to binary using repeated division method.

Solution.

		Remainders	
2	200		
2	100	0	LSB
2	50	0	
2	25	0	
2	12	1	
2	6	0	
2	3	0	
2	1	1	
	0	1	MSB

Reading the remainders from the bottom to the top, the result is : $200_{10} = 11001000_2$

On paper you may even compute the conversion as depicted through following example. (As you can see that this is just another way of representing the repeated division.)

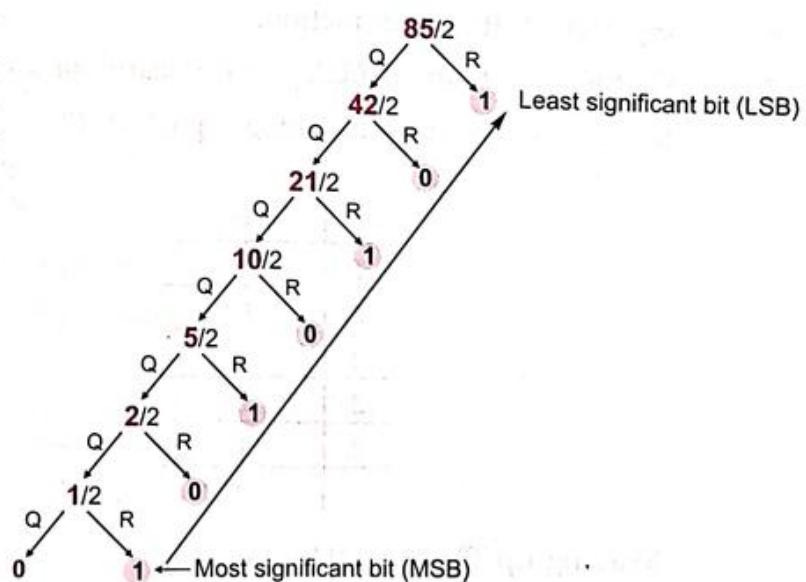
Example 1.5. Convert 85_{10} to binary using repeated division method.

Solution. (See on the right)

Q - Quotient and

R - Remainder

$$\therefore 85_{10} = 1010101_2$$



1.4.1A Converting Decimal Fractions to Binary

To convert a decimal fraction into binary, the procedure is to successively multiply the decimal fraction by the radix i.e., base (2 in this case) and collect all numbers to the left of the decimal point, reading down. Consider the following examples.

Example 1.6. Convert 0.375_{10} to binary.

Solution. Original number 0.375

Multiply (fractional part) 0.375 by 2

Multiply (fractional part) 0.75 by 2

Multiply (fractional part) 0.50 by 2

Integer Part
0
1
1

Write in this order

Reading down the integer parts, $0.375 = 0.011_2$

Example 1.7. Convert 0.54545_{10} to binary equivalent.

Solution. Original number 0.54545

$0.54545 \times 2 = 1.0909$

$0.0909 \times 2 = 0.1818$

$0.1818 \times 2 = 0.3636$

$0.3636 \times 2 = 0.7272$

$0.7272 \times 2 = 1.4544$

$0.4544 \times 2 = 0.9088$

$0.9088 \times 2 = 1.8176$

$0.8176 \times 2 = 1.6376$

$0.6376 \times 2 = 1.2704$

$0.2704 \times 2 = 0.5408$

Integer-Part

Writing order

This sequence keeps on continuing, thus, this particular number can never be expressed exactly in binary. Therefore, reading down the numbers, $0.54545 = 0.1000101110_2$ to 10 places. As the previous example illustrates, many decimal fractions cannot be represented as exact binary numbers.

To convert a mixed number such as 38.21 to binary, first convert the integer to its binary equivalent, then the fraction.

Example 1.8. Convert 38.21_{10} to its binary equivalent.

Solution. Converting the integer part i.e., 38.

		Remainders
2	38	
2	19	0
2	9	1
2	4	1
2	2	0
2	1	0
	0	1

Reading up, $38_{10} = 100110_2$

Converting the fractional part 0.21

		Integer-part
0.21 × 2	= 0.42	0
0.42 × 2	= 0.84	0
0.84 × 2	= 1.68	1
0.68 × 2	= 1.36	1
0.36 × 2	= 0.72	0
0.72 × 2	= 1.44	1
0.44 × 2	= 0.88	0
0.88 × 2	= 1.76	1
0.76 × 2	= 1.52	1
0.52 × 2	= 1.04	1
0.04 × 2	= 0.08	0

Reading down,

$$0.21_{10} = 0.00110101110$$

Adding the integer and fraction,

$$38.21_{10} = 100110.0011010111_2$$

Conversion Rules for Radix based Number Systems

1. Separate the integeral and the fractional parts.
2. For the integeral part, *divide by the target radix repeatedly*, and collect the successive remainders in reverse order.
3. For the fractional part, *multiply the fractional part by the target radix repeatedly*, and collect the successive integeral-parts in the same order.

1.4.2 Binary-to-Decimal Conversion

The binary number system is a positional system where each binary digit (bit) carries a certain weight based on its position relative to the LSB. Any binary number can be converted to its decimal equivalent simply by summing together the weights of the various positions in the binary number which contain a 1. To illustrate, consider a binary number 11011_2

1	1	0	1	1	₂
---	---	---	---	---	--------------

(binary)

Add positional weights
for all 1's $\rightarrow 2^4 + 2^3 + 0 + 2^1 + 2^0 = 16 + 8 + 2 + 1$
 $= 27_{10}$ (decimal)

Let's try another example with a greater number of bits i.e., 10110101_2

1	0	1	1	0	1	0	1	₂
---	---	---	---	---	---	---	---	--------------

Adding positional weights
of all 1's $\rightarrow 2^7 + 0 + 2^5 + 2^4 + 0 + 2^2 + 0 + 2^0 = 181_{10}$

Note that the procedure is to find the weights (*i.e.*, powers of 2) for each bit position that contains a 1, and then to add them up. Also note that the MSB has a weight of 2^7 even though it is the eighth bit; this is because the LSB is the first bit and has a weight of 2^0 .

The above method will always provide the correct decimal representation of a binary number. There is a second method, called the *dibble-dibble* method, that will also provide the solution. To use this method, start with the left-hand bit. Multiply this value by 2, and add the next bit to the right. Multiply the result value by 2, and add the next bit to the right. Stop when the binary point is reached. To illustrate,

11011_2 (binary)

Copy down the leftmost bit : 1

Multiply by 2, add next bit $(2 \times 1) + 1 = 3$

Multiply by 2, add next bit $(2 \times 3) + 0 = 6$

Multiply by 2, add next bit $(2 \times 6) + 1 = 13$

Multiply by 2, add next bit $(2 \times 13) + 1 = 27$ $\therefore 11011_2 = 27_{10}$

Let us try another example, 110100_2

The leftmost bit : 1

Multiply by 2, add next bit $(2 \times 1) + 1 = 3$

Multiply by 2, add next bit $(2 \times 3) + 0 = 6$

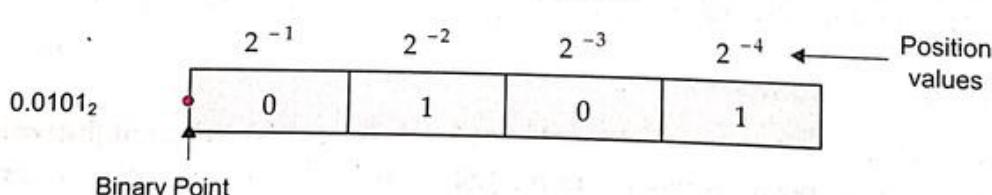
Multiply by 2, add next bit $(2 \times 6) + 1 = 13$

Multiply by 2, add next bit $(2 \times 13) + 0 = 26$

Multiply by 2, add next bit $(2 \times 26) + 0 = 52$ $\therefore 110100_2 = 52_{10}$

1.4.2A Converting Binary Fractions to Decimal

To find the decimal equivalent of binary fraction, take the sum of the products of each digit value (0 to 1) and its positional value. To illustrate :



$$\begin{aligned} &= (0 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4}) \\ &= 0 + 0.25 + 0 + 0.0625 = 0.3125 \end{aligned}$$

$$0.0101_2 = 0.3125_{10}.$$

Example 1.9. Convert 1101.000101_2 to decimal.

Solution.

$$\begin{aligned} &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 0 \times 2^{-5} + 1 \times 2^{-6} \\ &= 8 + 4 + 0 + 1 + 0 + 0 + \frac{1}{16} + 0 + \frac{1}{64} \\ &= 13 + 0.0625 + 0.015625 = 13.078125 \end{aligned}$$

$$1101.000101_2 = 13.078125_{10}$$

1.4.3 Decimal-to-Octal Conversion

A decimal integer can be converted to octal by using the same repeated-division method that we used in the decimal-to-binary conversion, but with a division factor of 8 instead of 2. An example is shown below :

		Remainders
8	266	
8	33	2
8	4	1
	0	4

Reading up, $266_{10} = 412_8$

Note that the first remainder becomes the least significant digit (LSD) of the octal number, and the last remainder becomes the most significant digit (MSD).

1.4.3A Converting Decimal Fraction to Octal

To convert a decimal fraction into octal, the procedure is to successively multiply the decimal fraction by the radix i.e., base (8 in this case) and collect all the numbers to the left of the decimal point, reading down. Consider the following examples.

Example 1.10. Convert 0.375_{10} to octal

Solution. Original number 0.375

$$\begin{array}{l} \text{Multiply (fraction part) } 0.375 \text{ by } 8 = 3.0 \quad 3 \\ \quad 0.375_{10} = 0.3_8 \end{array}$$

Example 1.11. Convert 0.015625_{10} to octal.

$$\begin{array}{l} \text{Solution. } 0.015625 \times 8 = 0.125 \quad 0 \\ \quad 0.125 \times 8 = 1.0 \quad 1 \downarrow \end{array}$$

Reading down, $0.015625_{10} = 0.01_8$

1.4.4 Octal-to-Decimal Conversion

An octal number, then, can be easily converted to its decimal equivalent by multiplying each octal digit by its positional weight. For example,

$$\begin{aligned} 372_8 &= 3 \times (8^2) + 7 \times (8^1) + 2 \times (8^0) \\ &= 3 \times 64 + 7 \times 8 + 2 \times 1 = 250_{10} \end{aligned}$$

Another example :

$$24.6_8 = 2 \times (8^1) + 4 \times (8^0) + 6 \times (8^{-1}) = 20.75_{10}$$

1.4.5 Octal-to-Binary Conversion

The primary advantage of the octal number system is the ease with which conversion can be made between binary and octal numbers. The conversion from octal to binary is performed by converting *each* octal digit to its 3-bit binary equivalent. The eight possible digits are converted as indicated in Table 1.3.

Table 1.3 *Binary equivalents of octal digits*

Octal Digit	0	1	2	3	4	5	6	7
Binary Equivalent	000	001	010	011	100	101	110	111

Using these conversions, any octal number is converted to binary by individually converting each digit.

For example, we can convert 472_8 to binary using 3 bits for each octal digit as follows :

$$\begin{array}{ccc} 4 & 7 & 2 \\ \downarrow & \downarrow & \downarrow \\ 100 & 111 & 010 \end{array}$$

Hence, octal 472 is equivalent to binary 100111010. As another example, consider converting 5431 to binary :

$$\begin{array}{cccc} 5 & 4 & 3 & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 101 & 100 & 011 & 001 \end{array}$$

Thus, $5431_8 = 101100011001_2$.

The same process applies equally on fractions. For example,

$$3.1_8 = \begin{array}{cccc} 3 & \bullet & 1 \\ \downarrow & & \downarrow \\ 011 & \bullet & 001 \end{array}$$

$3.1_8 = 011.001_2$

1.4.6 Binary-to-Octal Conversion

Converting from binary integers to octal integers is simply the reverse of the foregoing process. The bits of the binary integer are grouped into groups of *three* bits starting at the LSB. Then each group is converted to its octal equivalent (Table 1.3). To illustrate, consider the conversion of 100111010_2 to octal.

$$\begin{array}{ccc} 1 & 0 & 0 \\ \hline \downarrow & & \\ 4 & & \end{array} \quad \begin{array}{ccc} 1 & 1 & 1 \\ \hline \downarrow & & \\ 7 & & \end{array} \quad \begin{array}{ccc} 0 & 1 & 0 \\ \hline \downarrow & & \\ 2_8 & & \end{array}$$

Sometimes the binary number will not have even groups of 3 bits. For those cases, we can add one or two 0s to the left of MSB of the binary number to fill out the last group. This is illustrated below for the binary number 11010110.

0 added here to make it group of 3 bits



$$\begin{array}{ccc} 0 & 1 & 1 \\ \hline \downarrow & & \\ 3 & & \end{array} \quad \begin{array}{ccc} 0 & 1 & 0 \\ \hline \downarrow & & \\ 2 & & \end{array} \quad \begin{array}{ccc} 1 & 1 & 0 \\ \hline \downarrow & & \\ 6_8 & & \end{array}$$

Note that a 0 was placed to the left of the MSB in order to produce complete groups of 3.

The same process applies on fractions. But after the binary point, zeros are added to the right. For example

$$10110.0101_2 = \frac{0\ 1\ 0}{2} \quad \frac{1\ 1\ 0}{6} \quad \frac{0\ 1\ 0}{2} \quad \frac{1\ 0\ 0}{4}$$

2 zeros added here to
 make it a group of 3 bits

Note that, after the binary point, the groups of 3 bits are made starting from left-to-right. That is why, we added two zeros to make a group of three bits as the last group had only 1.

14.7 Decimal-to Hex Conversion

Recall that we did decimal-to-binary conversion using repeated division by 2, and decimal-to-octal using repeated division by 8. Likewise, decimal-to-hex conversion can be done using repeated division by 16. For example, to convert 423_{10} to hex,

Remainders

Reading up, $423_{10} = 1A7_{16}$

Similarly, to convert 214_{10} to hex,

Remainders		
16	214	
16	13	6
	0	D

Reading up, $214_{10} = D6_{16}$

Note that any remainders that are greater than 9 are represented by letters A through F.

1.4.7A Converting Decimal Fractions to Hex

The same old procedure applies here also, that is, successively multiply the decimal fraction by the radix (base) (16 here) and collect all the numbers to the left of the decimal point, reading down. For instance, 0.03125_{10} is converted as

$$\begin{array}{r}
 0.03125 \times 16 = 0.5 \\
 0.5 \times 16 = 8.0 \\
 0.03125_{10} = 0.08_{16}
 \end{array}$$

1.4.8 Hex-to-Decimal Conversion

A hex number can be converted to its decimal equivalent by using the fact that each hex digit position has a weight that is a power of 16. The LSD has a weight of $16^0 = 1$, the next higher

digit has a weight of $16^1 = 16$, the next higher digit has a weight of $16^2 = 256$, and so on. The conversion process is demonstrated in the examples below :

$$\begin{aligned}356_{16} &= 3 \times 16^2 + 5 \times 16^1 + 6 \times 16^0 \\&= 768 + 80 + 6 \\&= 854_{10}\end{aligned}$$

$$\begin{aligned}2AF_{16} &= 2 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 \\&= 512 + 160 + 15 \\&= 687_{10}\end{aligned}$$

Note that in the second example the value 10 was substituted for A and the value 16 for F in the conversion to decimal.

To convert a fractional number:

$$\begin{aligned}56.08_{16} &= 5 \times 16^1 + 6 \times 16^0 + 0 \times 16^{-1} + 8 \times 16^{-2} \\&= 80 + 6 + 0 + 8 / 256 = 86 + 0.03125 \\&= 86.03125_{10}\end{aligned}$$

1.4.9 Binary-to-Hex Conversion

Binary numbers can be easily converted to hexadecimal by grouping in groups of four starting at the binary point.

Example 1.12. Convert 1010111010_2 to hexadecimal.

Solution. **Group in fours** **10,1011,1010**
 Convert each number **2 B A**

Thus, the solution is 2 BA.

Example 1.13. Convert 10101110.010111, to hexadecimal.

Solution. Groups in fours (inserting zeros before MSB or in the end, wherever required)

1010, 1110 . 0101, 1100

A E . 5 C

$$10101110.010111_2 = AE.5C_{16}$$

1.4.10 Hex to Binary Conversion

Like the octal number system, the hexadecimal number system is used primarily as a "shorthand" method for representing binary numbers. It is a relatively simple matter to convert a hex number to binary. *Each* hex digit is converted to its 4-bit binary equivalent (Table 1.1). This is illustrated below for $9F2_{16}$.

$$9F2_{16} = \begin{array}{cccc} 9 & F & 2 \\ \downarrow & \downarrow & \downarrow \\ 1 & 0 & 0 & 1 \end{array} \quad \begin{array}{c} 1 \\ 1 \\ 1 \\ 1 \end{array} \quad \begin{array}{c} 0 \\ 0 \\ 1 \\ 0 \end{array}$$

$= 100111110010,$

Similarly, $3A6_{16} =$

3	A	6
\downarrow	\downarrow	\downarrow
0011	1010	0110

 $= 001110100110_2$

Note The hexadecimal and octal codes are used as shorthand means of expressing large binary numbers.

Consider another example

$3BF.5C_{16} =$

3	B	F	5	C
\downarrow	\downarrow	\downarrow	\downarrow	\downarrow
0011	1011	1111	0101	1100

 $= 001110111111.01011100_2$

Converting from Any Base to Any OTHER Base

As demonstrated in earlier examples and the table below, there is a direct correspondence between the number systems : with three binary digits corresponding to one octal digit ; four binary digits corresponding to one hexadecimal digit, which you can use to convert from one number system to another.

Table 1.4 Correspondence of binary and octal number systems

BIN	OCT	DEC
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7



For conversion from *base 2* to *base 8*, we use groups of three bits and vice-versa.

Table 1.5 Correspondence of Binary, Octal and Hexadecimal number systems.

BIN	HEX	OCT	DEC
0000	0	00	0
0001	1	01	1
0010	2	02	2
0011	3	03	3
0100	4	04	4
0101	5	05	5
0110	6	06	6
0111	7	07	7
1000	8	10	8
1001	9	11	9
1010	A	12	10
1011	B	13	11
1100	C	14	12
1101	D	15	13
1110	E	16	14
1111	F	17	15



For conversion from *base 2* to *base 16*, we use groups of four bits and vice-versa.

20

Let us consider some more examples regarding the same, i.e., converting from any number system to another.

Example 1.14. Convert $1948.B6_{16}$ to Binary and Octal equivalents.

Solution.

Hexadecimal	1	9	4	8	.	B	6
Binary	0001	1001	0100	1000	.	1011	0110

(By converting each individual Hex digit to equivalent 4 digit binary from above table 1.5)
 $\therefore 1948.B6_{16} = 0001100101001000.10110110_2$

New Octal number can be generated from above Binary equivalent i.e., as follows :

Binary	0	001	100	101	001	000	.	101	101	100
Octal	1	4	5	1	0	.	5	5	4	

(By creating groups of 3 binary digits and converting them into equivalent octal no.)
 $\therefore 1948.B6_{16} = 14510.554_8$

Example 1.15. Convert 75643.5704_8 to hexadecimal and binary numbers.

Solution. We shall convert it in following way :

- From octal to binary — by representing each octal digit to 3 digit binary number.
- From the complete binary number, we shall create groups of 4 binary digits around the decimal point.
- Convert each 4-digit-binary group to equivalent hex digit.

Octal	7	5	6	4	3	.	5	7	0	4
Binary	111	101	110	100	011	.	101	111	000	100

$$\therefore 75643.5704_8 = 111101110100011.101111000100_2$$

Binary	0111	1011	1010	0011	.	1011	1100	0100
Hexadecimal	7	B	A	3	.	B	C	4

$$\therefore 75643.5704_8 = 7BA3.BC4_{16}$$

After learning about various digital number systems, one must know how the data is represented in computers. Following sections illustrate the same.

1.5 Binary Representation of Numbers

While working with numbers, you have mainly worked with **integers** (e.g., +17, -23 etc.) or with **real numbers** (e.g., +17.015, -23.214 etc.)

Integers are *whole numbers* or *fixed-point numbers* with the radix point fixed after the least-significant bit. They are contrast to *real numbers* also called *floating-point numbers*, where the position of the radix point varies. It is important to take note that integers and floating-point

numbers are treated differently in computers. They have different representation and are processed differently (e.g., floating-point numbers are processed in a so-called floating-point processor). In this section, we shall explore how these two number types are represented in computers.

1.5.1 Binary Representation of Integers

To represent an integer, computers use a fixed number of bits. The commonly-used bit-lengths for integers are 8-bit, 16-bit, 32-bit or 64-bit. Besides bit-lengths, there are two representation schemes for integers :

- (i) *Unsigned Integers* can represent zero and positive integers.
- (ii) *Signed Integers* can represent zero, positive and negative integers.

1.5.1A Representing Unsigned Integers

An unsigned integer can be either a positive integer or zero but never a negative integer. Before we discuss how unsigned numbers are represented, consider a single digit decimal number :

- ◆ In a single decimal digit, you can write a number between 0 and 9 (maximum number is 9).
- ◆ In two decimal digits, you can write a number between 0 and 99 (maximum number is 99).
- ◆ In three decimal digits, you can write a number between 0 and 999 (maximum number is 999), and so on.

Since 9 is equivalent to $10^1 - 1$; 99 is equivalent to $10^2 - 1$; 999 is equivalent to $10^3 - 1$ etc., in n decimal digits, you can write a number between 0 and $10^n - 1$.

Analogously, in the binary number system,

an unsigned integer containing n bits can have a value between 0 and $2^n - 1$ (i.e., out of 2^n different values).

This fact is one of the most important and useful things to know about computers.

An n -bit pattern can represent 2^n distinct integers. An n -bit unsigned integer can represent integers from 0 to $2^n - 1$, as tabulated below :

Table 1.6 Ranges of Unsigned Integers represented through n -bits

n (bits)	Minimum value	Maximum Value
8	0	$2^8 - 1 (= 255)$
16	0	$2^{16} - 1 (= 65,535)$
32	0	$2^{32} - 1 (= 4,294,967,295)$ (9+ digits)
64	0	$2^{64} - 1 (= 18,446,744,073,709,551,615)$ (19+ digits)

1.5.1B Representing Signed Integers

Signed integers can represent zero, positive integers, as well as negative integers. Three representation schemes are available for signed integers :

- (i) Sign-Magnitude representation
- (ii) 1's Complement representation
- (iii) 2's Complement representation

In all the above three schemes, the most-significant bit (MSB) is called the *sign bit*.

I. Sign-Magnitude Representation

This is the conventional form for number representation. Integers are identified by their signs (+ or -) and a string of digits which represent the magnitude.

For example : +17 or 23 are positive integers (+ sign is frequently omitted)
 -14, -15 are negative numbers

To represent sign of a number (*i.e.*, + or -), utmost bit (called as *Most Significant Bit (MSB)*) is used. If it holds value 0, the sign is + and if holds 1, the sign is -.

That is, in sign-magnitude representation :

- ◆ The *most-significant bit* (MSB) is the *sign bit*, with value of 0 representing positive integer and value 1 representing negative integer.
- ◆ The remaining $n-1$ bits represent the *magnitude (absolute value)* of the integer. The absolute value of the integer is interpreted as "*the magnitude of the $(n-1)$ -bit binary pattern*".

For example, if in a computer, the word size is 1 byte (8 bits), then

- ◆ +15 will be represented as follows :

Number in binary notation

<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	0	0	0	1	1	1	1	Binary equivalent of 15 is 1111
0	0	0	0	1	1	1	1		
Most significant bit (MSB) (0 for + sign)	Least significant bit (LSB)								

- ◆ -23 will be represented as

<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>	1	0	0	1	0	1	1	1	Binary equivalent of 23 is 10111
1	0	0	1	0	1	1	1		
MSB (1 for - sign)	Number in binary notation								

Example 1.16. Given that $n=8$ and the sign-magnitude binary representation is $0100\ 0001_2$, determine the decimal integer represented.

Solution. Sign bit is 0 \Rightarrow positive number (+ sign)

$$\text{Absolute value is } 100\ 0001_2 = 2^6 + 2^0 = 64 + 1 = 65_{10}$$

Hence, the integer represented is $+65_{10}$.

Example 1.17. Given that $n=8$ and the sign-magnitude binary representation is $1\ 000\ 0001_2$, determine the decimal integer represented.

Solution. Sign bit is 1 \Rightarrow negative number (- sign)

$$\text{Absolute value is } 000\ 0001_2 = 2^0 = 1_{10}$$

Hence, the integer is -1_{10} .

Example 1.18. Given that $n=8$ and the sign-magnitude binary representation is $0\ 000\ 0000_2$, determine the decimal integer represented.

Solution.

Sign bit is 0 \Rightarrow positive number (+ sign)

Absolute value is $000\ 0000_2 = 0)_{10}$

Hence, the integer is $+0)_{10}$

Example 1.19. Given that $n=8$ and the sign-magnitude binary representation is $1\ 000\ 0000_2$, determine the decimal integer represented.

Solution.

Sign bit is 1 \Rightarrow negative number (- sign)

Absolute value is $000\ 0000_2 = 0)_{10}$

Hence, the integer is $-0)_{10}$

Total numbers which can be represented by N bit word, using sign and magnitude representation, are $2^N - 1$. In the above binary word of 8 bits, MSB is reserved for sign notation. Therefore, the maximum magnitude which can be represented is of 7 bits. Hence, maximum number which can be represented in 8 bits signed notation is $2^7 = 128$. An 8 bits word can represent total $2^8 - 1 = 255$ numbers i.e., -127 to 0 and 0 to +128. 16 bits (2 bytes) binary word can represent maximum $2^{15} = 32768$ numbers, one bit MSB is reserved for sign notation. And total ($2^{16} - 1 = 65535$) numbers it can represent are -32767 to 0 and 0 to +32768.

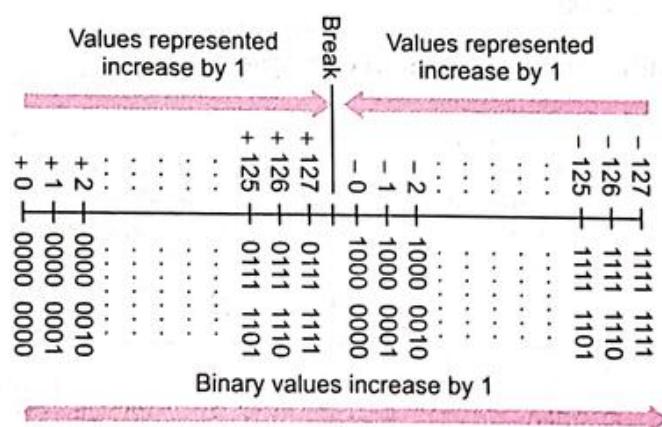


Figure 1.6 Sign-magnitude Representation.

Drawbacks/Problems Associated with Sign-magnitude Representation

As you must have noticed that this scheme allows you to write zero as two different numbers $+0$ or -0 , which is misleading and hence a drawback.

Major drawbacks of sign-magnitude representation are :

- There are two representations $0000\ 0000_2(+0)$ and $1000\ 0000_2(-0)$ for the number zero, which could lead to inefficiency and confusion.
- Positive and negative integers need to be processed separately.

II. One's Complement Representation

One's complement represents positive numbers by their binary equivalents (called *true forms*) and negative numbers by their 1's complements (called 1's complement forms).

For example, 1's complement of binary number 1001 will be 0110 and for 0011, it will be 1100.

Note To calculate 1's complement of a binary number, just replace every 0 with 1 and every 1 with 0.

Therefore + 6 and - 6 will be represented as 0110 (binary equivalent of 6) and 1001 (1's complement of 6) respective. Total numbers which can be represented using 1's complement are $2^N - 1$ where N is the word size (number of bits in a word). Therefore, an 8 bit word can represent maximum $2^8 - 1 = 255$ numbers. 1's complement can represent + 0(0000) and - 0(1111), but there is no significance of + 0 0 or - 0, therefore, only 0000 is used to represent 0 (zero).

Example 1.20. Find the one's complement form of - 13.

Solution. $+13 = 0000\ 1101$
 $-13 = 1111\ 0010$

This system is called the *one's complement* because the number can be subtracted from 1111 1111 to obtain the result. The answer will be the same as if all the 1's were changed to 0's and all the 0's to 1's.

Example 1.21. Find the one's complement representation of - 13.

Solution.
$$\begin{array}{r} 1111\ 1111 \\ - 0000\ 1101 \\ \hline 1111\ 0010 \end{array}$$

Note that this is the same result as in example 1.14.

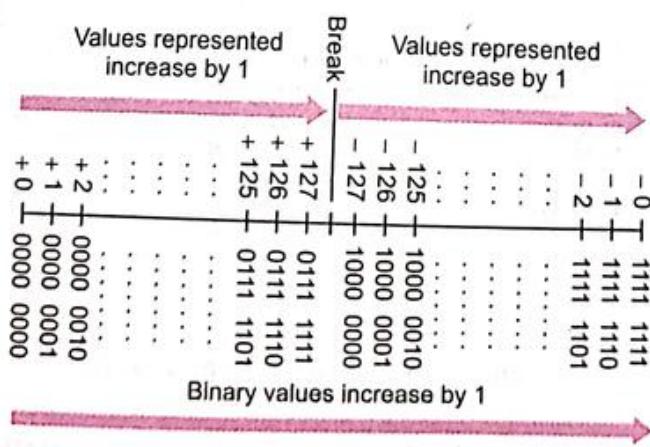


Figure 1.7 1's Complement Representation.

Drawbacks/Limitations of One's Complement Notation

The drawbacks/limitations of one's complement representation scheme are :

- There are two representations $0000\ 0000_2$ and $1111\ 1111_2$ for zero.
- The positive integers and negative integers need to be processed separately.

III. Two's Complement Representation

Two's complement method represents positive numbers in their true forms i.e., their binary equivalents and negative numbers in 2's complement form.

For instance, number 6 (0110)'s 2's complement will be calculated as follows :

$$\text{1's complement of } 0110 = 1001$$

$$\text{2's complement of } 0110 = \begin{array}{r} +1 \\ \hline 1010 \end{array}$$

(Binary addition rules are $0+0=0$; $0+1=1$; $1+0=1$; $1+1=10$; $1+1+1=11$)

Therefore numbers 6 and -6 will be represented as 0110 (binary equivalent of 6) and 1010 (2's complement of 6).

Total numbers which a word of N bits can present, using 2's complement representation, are 2^N .

Therefore, a 4 bit word can represent $2^4 = 16$ numbers.

Example 1.22. Express -4 in two's complement form.

Solution.	Positive expression of number	0000 0100
	Complement	1111 1011
	Add 1	1111 1100

Therefore, -4 equals 1111 1100.

Example 1.23. Express -17 in two's complement form.

Solution.	Positive expression of number	0001 0001
	Complement	1110 1110
	Add 1	1110 1111

Another method of converting a positive number to its two's complement is as follows :

Starting at the least significant bit and working from right to left, copy each bit down up to and including the first 1 bit encountered. Then complement the remaining bits.

Example 1.24. Express -4 in two's complement form.

Solution.	Original number	0000 0100
	Copy up to first 1	100
	Complement remaining bits	1111 1100

Compare this result with Example 1.22.

Example 1.25. Express -17 in two's complement form.

Solution.	Original number	0001 0001
	Copy up to first 1	1
	Complement remaining bits	1110 1111

Compare this result with Example 1.23.

Note 2's complement of a number is calculated by adding 1 to its 1's complement.

Two's complement numbers have some very interesting characteristics. Consider a four-bit word length. The decimal equivalents would be as shown in Table 1.7.

Table 1.7 Two's Complement Numbers

Binary	Decimal
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

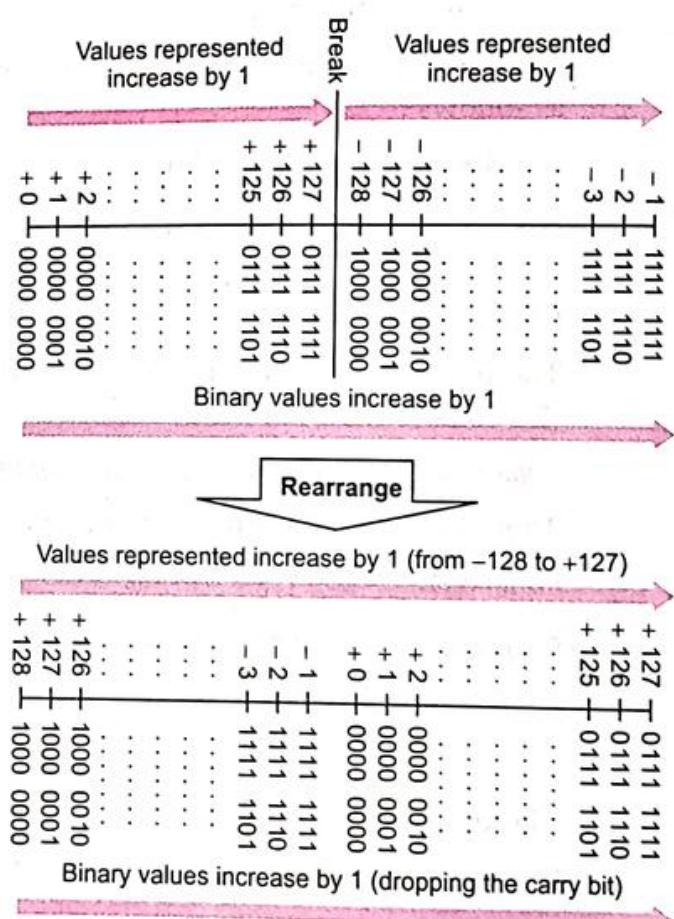


Figure 1.8 2's Complement Representation

Looking at Table 1.7 and Fig. 1.8, you can observe the following :

- (i) There is one unique 0.
- (ii) The two's complement of 0 is 0.
- (iii) The leftmost bit cannot be used to express quantity. It is a sign bit such that if it is a '1', the number is negative, and if it is a '0', the number is positive.
- (iv) There are eight negative integers, seven positive integers, and one 0, making a total of 16 unique states. In the two's complement system, there will always be $2^{P-1} - 1$ positive integers, 2^{P-1} negative integers, and one 0, for a total of 2^P unique states, where P is the number of bits in the binary word.
- (v) Significant information is contained in the 1's of the positive numbers and the 0's of the negative numbers. Think about that for a while.
(Hint : Consider -2 and +2 in a 16-bit number.)
- (vi) To convert a negative number to a positive number, you just need to find its two's complement.

Generally, the negative numbers are stored in 2's complement form and positive numbers in sign and magnitude form.

Computers use 2's Complement Representation for Signed Integers

Out of the three representations for signed integers : *sign-magnitude*, *1's complement* and *2's complement*, computers use 2's complement in representing signed integers. This is because :

- There is only one representation for the number zero in 2's complement, instead of two representations in sign-magnitude and 1's complement.
- Positive and negative integers can be treated together in addition and subtraction. Subtraction can be carried out using the "addition logic".

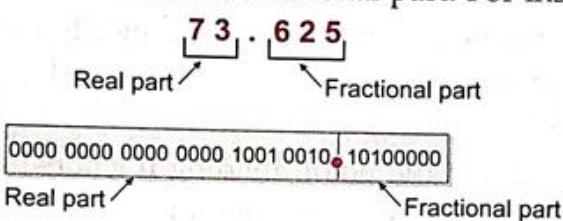
(This point will become clear to you in a later section where we shall discuss addition and subtraction with different representation schemes.)

1.6 Floating-Point Representation (For Real Numbers)

Floating-point representation is used to represent real numbers, i.e., numbers with fractions. A floating-point representation scheme allows you to store non-fractional and fractional parts separately. In this section, we shall explore different schemes for floating-point representation.

1.6.1 Fixed-Point Representation

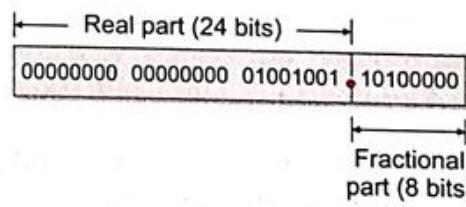
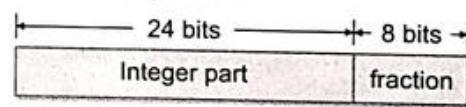
In fixed-point representation scheme, real numbers are represented in binary form and there are some bits reserved for the real and fractional part. For instance,



The **fixed-point representation** splits the bits of the representation between the places to the left of the decimal point and places to the right of the decimal point.

For example, a 32-bit fixed-point representation might allocate 24 bits for the integer part and 8 bits for the fractional part.

To represent 73.625, we would use the first 24 bits to store 73 in binary, and we'd use the remaining 8 bits to hold value 0.625 (in binary). Thus, fixed-point 32-bit representation for number 73.625 would be as shown in the figure on the right.



Note Fixed-point representation approach has a limitation that very large numbers or very small fractions cannot be represented.

Fixed-point representation works reasonably well as long as you work with numbers within the supported range. For instance, the 32-bit fixed-point representation described above can represent any multiple of $1/256$ from 0 up to $2^{24} \approx 16.7$ million. If your program uses numbers within this range only, you can safely work with this scheme, but programs frequently need to work with numbers from a much broader range. For this reason, fixed-point representation isn't used very often in today's computing world.

1.6.2 Scientific Notation (Exponential Notation)

Exponential Notation or Scientific notation is a conventional method for representing floating-point numbers. As you must be aware that scientific notation has the following configuration:

$$350.5 = 3.505 \times 10^2$$

where 3 is the non-fractional part and 0.505 is the fractional part ; 10 is the radix or base of the system and 2 is the exponent to the radix. As this number is a Decimal number, the base is 10.

Following figure (1.9) illustrates the components of the scientific notation in Decimal number system as well as in Binary Number System.

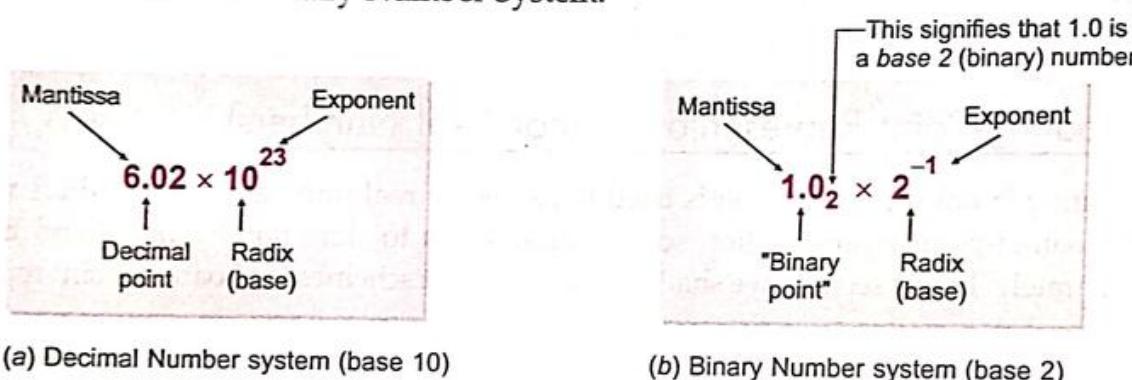


Figure 1.9 Components in Scientific notation

Thus, you see that a floating-point number is typically expressed in the scientific notation, with a *fraction* (F) and an exponent (E) of a certain radix (r) in the form of $F \times r^E$. Decimal numbers use radix of 10($F \times 10^E$) ; while binary numbers use radix of 2($F \times 2^E$).

You must have noticed that the point dividing fractional and non-fractional parts is called *decimal point* in decimal numbers and *binary point* in binary numbers. In general, you can call this point as *radix point*.

Please remember here that the same number can be represented through various mantissa and exponents, e.g., number 350.56×10^2 is equivalent to all the following numbers :

$$35.056 \times 10^3 \quad 3.5056 \times 10^4 \quad 3505.6 \times 10^1$$

Similarly, binary number 1011.001×2^2 is equivalent to all the following numbers :

$$101.1001 \times 2^3 \quad 10.11001 \times 2^4 \quad 1.011001 \times 2^5$$

1.6.3 Normalized Scientific Notation

Representation of floating point number is not unique. For example, the number 55.66 can be represented as 5.566×10^1 , 0.5566×10^2 , 0.05566×10^3 , and so on. The fractional part can be *normalized*, however. In the normalized form, there is only a single non-zero digit before the radix point. For example, *decimal number* 123.4567 can be normalized as 1.234567×10^2 ; *binary number* 1010.1011 can be normalized as 1.0101011×2^3 .

Here carefully notice one thing that in binary number system, the normalized numbers will always have binary value 1 before the binary point – as per the condition that there should be only a single non-zero number before binary point. (As there is only single bit 1 that is non-zero in binary number system)

Note In the normalized scientific form, there is only a single non-zero digit before the radix point.

1.6.4 Mantissa-Exponent Notation

Computers mostly represent real numbers in a form known as **Mantissa and Exponent form**. Similar to scientific notation, the value of a floating-point binary number is represented in the form :

Mantissa $\times 2^{\text{exponent}}$

where mantissa is in normalised binary form. Let us understand it with the help of an example. Suppose you have to represent decimal number 18.5 in Mantissa-Exponent Notation.

Step 1 Convert 18.5 into binary which is : 10010.1.

Step 2 Normalize this binary number (so that there is only one non-zero bit is on the left of binary-point.)

For this we need to move the binary point to the left by four places, i.e., now the number is like :

$$1.00101 \times 2^4 \rightarrow \text{this } 4 \text{ is in decimal number}$$

Step 3 Represent everything in binary form. The power or exponent 4 is 100 in binary i.e., the number is like :

$$1.00101 \times 2^{100} \xrightarrow{(binary\ equivalent\ of\ 4)}$$

Step 4 So, we now have Mantissa as :

1.00101

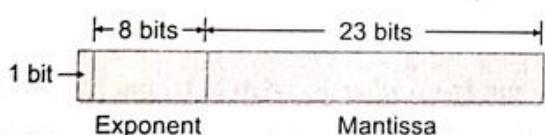
and exponent as 100

Mantissa is the part of a floating-point number which represents the significant digits of that number.

Exponent represents the power to base to be multiplied with mantissa to get the actual floating point number.

Representation in Memory

IEEE standard way of representing floating-point numbers with mantissa and exponent scheme is as follows :



A 32-bit representation has :

Leftmost 1 bit as sign bit

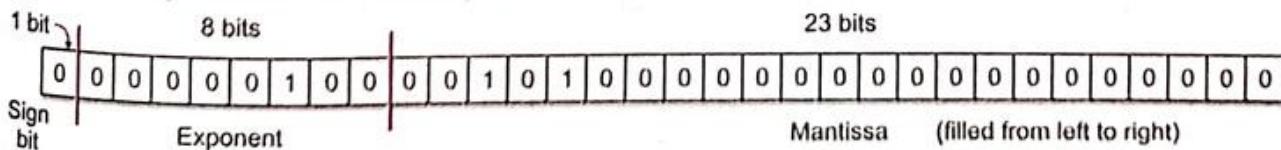
next 8 bits as exponent

next 23 bits as Mantissa.

Here you need to know about one important thing and which is that the bit before binary point is not stored *i.e.*, from the Mantissa computed above 1.00101, only the fractional part .00101 will be stored. The reason being since in the normalized form, the mantissa must have exactly one nonzero bit to the left of its binary point, and the only nonzero bit is 1, we know that the bit to the left of the binary point must be a 1. There's no point in wasting space in inserting this 1 into our bit pattern, so computer includes only the bits of the mantissa to the right of the binary point.

So the number computed above : 1.00101×2^{100}

will be represented in memory as



Let us convert above floating-point number into decimal form :

Sign bit = 0 → positive number

Exponent = $00000100_2 = 4_{10}$

Mantissa = 00101

⇒ it represents 1.00101 as bit 1 to the left of binary is not stored as it is always 1.

1.00101×2^4 is the number = 10010.1

$$\text{Non-fractional part} = 10010 = 1 \times 2^4 + 1 \times 2^1 = 10 + 2 = 18$$

$$\text{Fractional Part} \quad 1 \times 2^{-1} = \frac{1}{2} = 0.5$$

$$\therefore 10010.1_2 = 18.5_{10}$$

Excess Notation

Many computers also represent floating point number in **Excess-notations**, which helps in representing numbers in a sorted order.

The excess notation is a means of representing both negative and positive numbers in a manner in which the order of the bit patterns is maintained. Let us understand it with the help of an example :

Consider the 8 bit-patterns in 3 bits (in decreasing order)

111 110 101 100 011 011 001 000

These bit patterns when interpreted as natural numbers represent

7 6 5 4 3 2 1 0

When interpreted as integers in 2's complement represent :

-1 -2 -3 -4 3 2 1 0

Now in excess notation a fixed number called *magic number* is added to each number before representing it in binary form. Say if the magic number is 4, this becomes Excess-4 notation and will represent numbers as follows :

Original number	Excess 4	Binary notation
0	$4(0+4)$	100
1	$5(1+4)$	101
2	$6(2+4)$	110
3	$7(3+4)$	111
-1	$3(-1+4)$	011
-2	$2(-2+4)$	010
-3	$1(-3+4)$	001
-4	$0(-4+4)$	000

Compare it with earlier representations, we have

Bit-pattern	National numbers	2's complement notation	Excess-4 notation
111	7	-1	3
110	6	-2	2
101	5	-3	1
100	4	-4	0
011	3	3	-1
010	2	2	-2
001	1	1	-3
000	0	0	-4

Sorted Signed Numbers

As you can see that excess notation is capable of representing signed numbers in sorted order, hence excess notations are often used by computers.

Consider the following examples that represent numbers in Excess-7 notation.

Note Excess notation is also sometimes referred to as *Biased notation*.

Example 1.26. Represent $-96_{(10)}$ in Excess-7 notation.

Solution.

- (a) First we convert our desired number to binary : $-1100000_{(2)}$.
- (b) Then we convert this to binary scientific notation : $-1.100000_{(2)} \times 2^6$.
- (c) Then we fit this into the bits.
 - (i) We choose 1 for the sign bit since the number is negative.
 - (ii) We add 7 to the exponent and place the result into the four exponent bits. For this example, we arrive at $6 + 7 = 13_{(10)} = 1101_{(2)}$.
 - (iii) The three mantissa bits are the first three bits following the leading 1: 100.

Thus we end up with 1 1101 100.

Example 1.27. Decode the number 0 0101 100 represented in Excess-7 notation.

Solution.

- (i) We observe that the number is positive, and the exponent bits represent $0101_{(2)} = 5_{(10)}$.
- (ii) This is 7 more than the actual exponent, and so the actual exponent must be $5 - 7 = -2$. Thus, in binary scientific notation, we have $1.100_{(2)} \times 2^{-2}$.
- (iii) We convert this to binary $1.100_{(2)} \times 2^{-2} = 0.011_{(2)}$.
- (iv) We convert the binary into decimal

$$0.011_{(2)} = \frac{1}{4} + \frac{1}{8} = \frac{3}{8} = 0.375_{(10)}$$

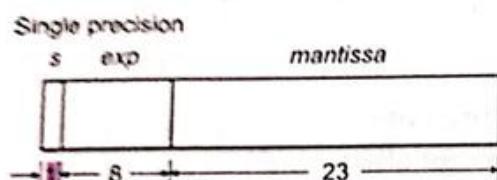
1.6.5 Single and Double Precision Representation Schemes

In computers, floating-point numbers are represented in scientific notation of fraction (F) and exponent (E) with a radix of 2, in the form of $F \times 2^E$. Both E and F can be positive as well as negative. Modern computers adopt IEEE 754 standard for representing floating-point numbers.

There are two representation schemes : 32-bit single-precision and 64-bit double-precision.

- ◆ Single precision numbers include an 8-bit exponent field and a 23-bit fraction, for a total of 32 bits.
 - ◆ Double precision numbers have an 11-bit exponent field and a 52-bit fraction, for a total of 64 bits.

Figure 1.10 illustrates these two representation schemes.



Note Mantissa is also sometimes referred to as significand.

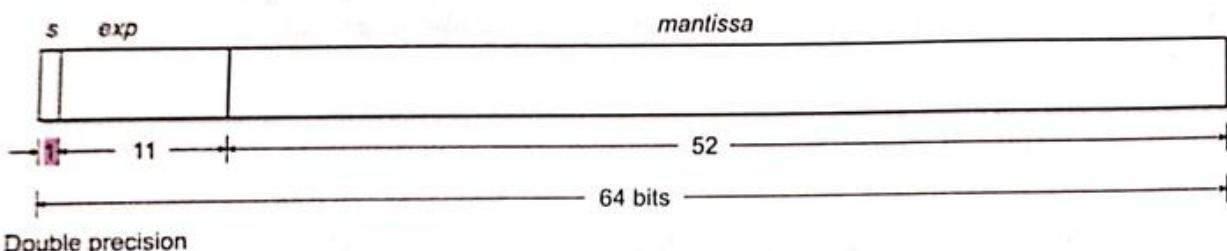


Figure 1.10 Single and double-precision number representation

Example 1.28. What is the decimal value of this single Precision float ? (Excess/Bias 127)

Solution. Sign = 1 is negative

$$\text{Exponent} = (01111100)_2 = 124,$$

Adjust bias by subtracting excess = 124 - 127 = -3

$$\text{Significand} = (1.0100 \dots 0)_2 = 1 + 2^{-2} = 1.25 \quad (1. \text{ is implicit})$$

Value in decimal $\equiv -1.25 \times 2^{-3} \equiv -0.15625$

Example 1.29 What is the decimal value of following binary number ? (Excess/Bias 127)

Solution.

implicit

Adjusting excess/bias

$$\begin{aligned}\text{Value in decimal} &= + (1.01001100 \dots 0)_2 \times 2^{130-127} \\ &= (1.01001100 \dots 0)_2 \times 2^3 = (1010.01100 \dots 0)_2 = 10.375\end{aligned}$$

1.6.5A Trade-off between Size of Mantissa and Exponent

As you know that the maximum number represented by mantissa depends upon the number of bits allotted for mantissa ; more the bits, more the accuracy (*i.e.*, precision). It is possible to improve the accuracy of a floating point number (*i.e.*, the range of numbers held can be increased) by increasing the number of bits devoted to the mantissa. But increasing the mantissa-size would reduce the exponent size, thereby reducing the range of numbers that can be represented. Hence, there will always be a trade-off between Precision/accuracy and Range when storing real numbers using floating point notation, as there will always be a set number of bits allocated to storing real numbers with the potential to increase or decrease the number of bits used for the mantissa against the number of bits used for the exponent.

- ❖ The more digits assigned for the mantissa, the higher the precision and lower the range.
- ❖ The more digits assigned for the exponent the higher the range and lower the precision.

Note The higher the number-of-bits for mantissa, the higher the precision and lower the range and vice versa.

1.7 Binary Arithmetic

After learning to represent numbers in various number systems, let us now discuss binary arithmetic. In the following lines, we shall be discussing various methods used to perform arithmetic operations on binary numbers.

1.7.1 Binary Addition

ALUs don't directly work upon decimal numbers ; rather they process binary numbers as a computer can understand only binary numbers. There are *five* basic cases for binary addition that must be understood before going on. These are :

$$\text{Case 1 : } 0 + 0 = 0$$

i.e., addition of two binary 0's (zero) results into a binary 0 (zero).

$$\text{Case 2 : } 0 + 1 = 1$$

i.e., addition of a binary 0 (zero) and a binary 1 (one) results into binary 1 (one).

$$\text{Case 3 : } 1 + 0 = 1$$

i.e., addition of a binary 1 (one) and a binary 0 (zero) results into binary 1 (one).

$$\text{Case 4 : } 1 + 1 = 10$$

i.e., Binary 1 + Binary 1 equals Binary 10. Or we can say that in binary numbers, one plus one equals zero (0), carry one (1).

$$\text{Case 5 : } 1 + 1 + 1 = 11$$

i.e., Binary 1 + Binary 1 + Binary 1 equals Binary 1, carry Binary 1.

Let us summarize these rules

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

$$1 + 1 + 1 = 11$$

[0 with carry 1]

[1 with carry 1]

Let us now perform binary addition on bigger binary numbers. The binary numbers are also added column-by-column just like decimal numbers. Also, the way results larger than largest decimal digit are carried-over, in binary addition, results larger than 1 are also carried over.

For example,

Decimal Addition

$$\begin{array}{r}
 & 1 \leftarrow \text{This is carried over as } 5+7=12 \text{ which is more than 9,} \\
 & \text{hence carried over.} \\
 5 & \\
 + 7 & \\
 \hline
 12 & \\
 \end{array}$$

Similarly, Binary Addition will be

$$\begin{array}{r}
 1111 \quad \text{Carries} \\
 + 1101 \quad \text{(Equivalent of } 15_{10}) \\
 \hline
 11011 \quad \text{(Equivalent of } 27_{10}) \\
 + 101010 \quad \text{(Equivalent of } 42_{10}) \\
 \hline
 \end{array}$$

Let us perform another addition, step-by-step for

$$\begin{array}{r}
 11100 \\
 + 11010 \\
 \hline
 \end{array}$$

Start with the least significant column i.e., the right most column to get

$$\begin{array}{r}
 11100 \\
 + 11010 \\
 \hline
 0
 \end{array}$$

In all examples, $0+0$ gives 0.

Next, add the bits of the second column (second from right) as follows :

$$\begin{array}{r}
 11100 \\
 + 11010 \\
 \hline
 10
 \end{array}$$

This time, $0+1$ gives

The addition in third column gives

$$\begin{array}{r}
 11100 \\
 + 11010 \\
 \hline
 110
 \end{array}$$

In this case, $1+0$ results in

$$\begin{array}{r}
 11100 \\
 + 11010 \\
 \hline
 0110 \quad (\text{carry 1})
 \end{array}$$

As you see, $1 + 1$ produces 10 i.e., 0 with a carry of 1.

Finally, the last column gives

$$\begin{array}{r}
 & 1 \xleftarrow{\text{Carries}} \\
 & 11100 \\
 + & 11010 \\
 \hline
 & 110110
 \end{array}$$

Here, $1 + 1 + 1$ (previously generated carry) results in 11, recorded as 1 with a carry to the next higher column.

Example 1.30. Add the binary numbers 01010111 and 00110101.

Solution. If you add the bits column by column as earlier explained you will get

$$\begin{array}{r}
 & 1111111 \xleftarrow{\text{Carries}} \\
 & 01010111 \\
 + & 00110101 \\
 \hline
 & 10001100
 \end{array}$$

Example 1.31. Add the binary numbers 1011 and 110.

Solution.

$$\begin{array}{r}
 & 11 \xleftarrow{\text{Carries}} \\
 & 1011 \\
 + & 110 \\
 \hline
 & 10001
 \end{array}$$

Example 1.32. Add binary numbers 11110 and 11.

Solution.

$$\begin{array}{r}
 & 111 \xleftarrow{\text{Carries}} \\
 & 11110 \\
 + & 11 \\
 \hline
 & 100001
 \end{array}$$

Example 1.33. Add binary numbers 11.01 and 101.11.

Solution.

$$\begin{array}{r}
 & 1111 \xleftarrow{\text{Carries}} \\
 & 11.01 \\
 + & 101.11 \\
 \hline
 & 1001.00
 \end{array}$$

1.7.2 Binary Subtraction

Binary subtraction is performed in similar manner as that of decimal subtraction. Because the binary number system has only two digits, binary subtraction requires more borrowing operations than decimal number system.

To subtract binary numbers, we need to discuss four cases :

Case 1: $0 - 0 = 0$

Case 2: $1 - 0 = 1$

Case 3: $1 - 1 = 0$

Case 4: $1 - 1 = 1$ i.e., $0 - 1 = 1$ with a borrow of 1.

The last result is representation of

$$10 - 1 = 1$$

which makes sense (because $1 + 1 = 10$).

To subtract larger binary numbers, one needs to subtract column by column, borrowing from the next higher column whenever necessary. For example, in subtracting 101 from 111, one should proceed like this :

$$\begin{array}{r} 111 \\ - 101 \\ \hline 010 \end{array}$$

Starting on the right, $1 - 1$ gives 0 ; then $1 - 0$ is 1 ; finally, $1 - 1$ is 0.

Notice that nothing is borrowed in this example. But in the next example, you'll see "borrow" taking place.

Example 1.34. Subtract 1 (one) from 100_2 .

Solution.

$$\begin{array}{r} & 1 & \leftarrow \text{Borrows} \\ & 1 & \\ 1 & 0 & 0 \\ & \underline{-} & 1 \\ & 1 & 1 \end{array}$$

Since, in the last most column, a 1 cannot be subtracted from a 0, a 1 must be borrowed from the next (i.e., 2nd column from right) column. But it is also 0, so a 1 must be borrowed from the next (i.e., 3rd from right) column, making 3rd column 0 and the 2nd column 10. A 1 can now be borrowed from the 2nd column, making it a 1 and the first column 10.

$$10 - 1 = 1 \text{ in the 1st column}$$

$$1 - 0 = 1 \text{ in the 2nd column}$$

$$0 - 0 = 0 \text{ in the 3rd column}$$

Therefore, $100_2 - 1_2 = 11_2$.

Example 1.35. Subtract 101 from 1001. Both of these numbers are binary numbers.

Solution.

$$\begin{array}{r} & 1 \\ & \swarrow \\ & 1 & 0 & 0 & 1 \\ \text{Borrows} & & & & \\ & & 1 & 0 & 1 \\ & \underline{-} & & & \\ & & 1 & 0 & 0 \end{array}$$

Example 1.36. Subtract 11_2 from 10000_2 .

Solution.

$$\begin{array}{r} & 1 & 1 & 1 & 1 \\ & \swarrow \\ & 1 & 0 & 0 & 0 & 0 \\ \text{Borrows} & & & & & \\ & & & 1 & 1 \\ & \underline{-} & & & \\ & & 1 & 1 & 0 & 1 \end{array}$$

Example 1.37. Subtract 100.1_2 from 110.01_2 .

Solution.

$$\begin{array}{r} 110.01 \\ - 100.1 \\ \hline 1.11 \end{array}$$

Issues with Addition/Subtraction in Sign and Magnitude System

Recall that Sign & Magnitude representation system represents both positive and negative integers using just 1's and 0's – by using the most significant bit (msb), x_{N-1} , to designate the sign and the remaining ($N - 1$) bits to designate the magnitude (for N -bit capacity).

That is :

$\underbrace{x_{N-1}}_{\text{sign}} \quad \underbrace{x_{N-2} \dots x_1}_{\text{magnitude}} \quad x_0$ and the usual convention is

$$x_{N-1} = 0 \Rightarrow X \text{ is positive}$$

$$x_{N-1} = 1 \Rightarrow X \text{ is negative}$$

$$+6 \rightarrow 0110$$

$$-5 \rightarrow 1101$$

While the *Sign and magnitude* format is intrinsically simple, it leads to significant complications if the normal rules of arithmetic are applied.

If you use this system for addition or subtraction, this might give you correct or incorrect results. *It gives correct result if both numbers are of the same sign and the result very well fits in $N - 1$ bits. In other cases, it gives incorrect results.* Have a look at examples below that highlight this problem.

Examples 1.38. Add numbers +3 and +2 using sign and magnitude representation.

Solution.

$$\begin{array}{r} (+3) \quad 0011 \\ \text{add} \quad (+2) \quad 0010 \\ \hline 0101 \end{array} \rightarrow (+5) \text{ is correct}$$

Examples 1.39. Add numbers +3 and -2 using sign and magnitude representation.

Solution.

$$\begin{array}{r} (+3) \quad 0011 \\ \text{add} \quad (-2) \quad 1010 \\ \hline 1101 \end{array} \rightarrow (-5) \text{ is incorrect}$$

Because of the problems arising with addition and subtraction, the sign and magnitude system is not generally used where arithmetic functions are required; it may, however, be used for data storage.

Note

Because of the problems arising with addition and subtraction, the sign and magnitude system is not generally used where arithmetic functions are required; it may, however, be used for data storage.

1.7.3 Binary Addition and Subtraction using One's Complement Notation

You already have learnt about 1's complement and 2's complement notations. Though 2's complement arithmetic is used by most computers, yet some use 1's complement also. This section explains to you the binary operations in 1's complement arithmetic.

As you know that a negative number can be represented in 1's complement by changing all 0's to 1's and vice-versa if the positive binary number e.g., +13 is represented as 0000 1101 and -13 as 11110010, which is 1's complement of +13.

If a binary number, say b is to be subtracted from another binary number, say a , then it may also be interpreted as $a + (-b)$ in place of $a - b$ i.e., it becomes a special case of addition, wherein, complement is added.

In 1's complement arithmetic, the two numbers are added including the sign bits. Let us learn various cases possible in 1's complement arithmetic.

Case 1 Positive number added to another positive number. For example,

$$\begin{array}{r}
 & \text{Sign bit} \rightarrow 11 \\
 +7 & \\
 +6 & \\
 \hline
 13 & 0,0110 \\
 & \underline{0,1101} \\
 & 0,1101
 \end{array}$$

This type of addition is straightforward.

Case 2 Addition of a Positive and Negative number.

Case 2.1 Positive number with a greater magnitude added to a negative number with lesser magnitude. For example,

$$\begin{array}{r}
 & \text{Sign bit} \rightarrow 11 \\
 +7 & 0,0111 \\
 -3 & \underline{1,1100} \\
 \hline
 4 & 10,0011 \\
 & \underline{\quad\quad\quad\quad\quad} \\
 & 1 \quad \text{Add overflow bit to LSB} \\
 & \underline{0,0100} \quad \text{Result}
 \end{array}$$

1's complement of 3 representing -3

See an overflow is generated, notice an extra bit beyond sign bit. This overflow bit is added to the least significant bit of the sum to get the correct result. This process is called *end around carry*.

Case 2.2 Negative number has greater magnitude than positive number, e.g.,

$$\begin{array}{r}
 & \text{Sign bit} \rightarrow 1,1000 \\
 -7 & 1's complement of 7 representing -7 \\
 +3 & 0,0011 \\
 \hline
 -4 & \underline{1,1011} \quad (\text{this is 1's complement of 4, depicting } -4)
 \end{array}$$

Notice that no overflow is generated. This answer is correct as the sign bit is 1 and the result is in 1's complement form.

Case 3 Addition of two negative numbers, e.g.,

$$\begin{array}{r}
 & 1 \\
 & 1,1000 \\
 -7 & \underline{1,1100} \\
 -3 & \underline{11,0100} \\
 \hline
 -10 & \underline{\quad\quad\quad\quad\quad} \\
 & 1 \quad (1's complement representing -10)
 \end{array}$$

Notice that an overflow is generated, this overflow is added to the least significant bit. If the sign bit is 1 depicting the negative result, then the result is correct otherwise incorrect e.g.,

$$\begin{array}{r}
 \text{Sign bit} \rightarrow 1,0111 \\
 -8 \\
 -9 \\
 \hline
 -17 \\
 \hline
 0,1110 \quad (\text{Incorrect sign bit, thus incorrect result})
 \end{array}$$

In such case bigger unit should be used for addition. For instance, in the previous case the unit used is 4 bit number which can represent a number with magnitude upto 15. Bigger numbers will definitely cause incorrectness. Now let us do the same example with 8 bits.

$$\begin{array}{r}
 \text{Sign bit} \rightarrow 1\ 1\ 1\ 1\ 1\ 1\ 1 \\
 1,1110111 \\
 -8 \\
 -9 \\
 \hline
 -17 \\
 \hline
 1,1101110 \quad (1\text{'s complement representing } -17)
 \end{array}$$

[Notice the result is 1's complement of 0,0010001, which is binary equivalent of +17]

1.7.4 Binary Addition and Subtraction using 2's complement Notation

You have learnt that to represent a negative number, its 2's complement of a number is calculated by adding 1 to the least significant bit of its 1's complement. Let us now discuss the addition and subtraction mechanisms using 2's complement notation.

Case 1 Two positive numbers are added.

This situation is identical to the same case in 1's complement notation e.g.,

$$\begin{array}{r}
 \text{Sign bit} \rightarrow 1\ 1\ 1 \\
 +5 \\
 +3 \\
 \hline
 8 \\
 \hline
 0,1000
 \end{array}$$

Case 2 Addition of a positive and negative number (i.e., subtraction of a number from another number).

Case 2.1 Positive Number has greater magnitude.

$$\begin{array}{r}
 \text{Sign bit} \rightarrow 0,0111 \\
 +7 \\
 -3 \\
 \hline
 4 \\
 \hline
 1\ 0,0100
 \end{array}$$

Ignore this overflow

The result is 0,0100 which is correct.

The rule in this case is that after adding the two numbers, ignore the overflow if any. The result should be positive for the above example, the sign bit is zero, which is correct.

Case 2.2 Negative number has greater magnitude.

$$\begin{array}{r}
 -7 \\
 +3 \\
 \hline
 -4
 \end{array}
 \quad \text{Sign bit} \rightarrow 1,1001$$

$$\begin{array}{r}
 0,0011 \\
 \hline
 1,1100
 \end{array}
 \quad (2\text{'s complement representing } -4)$$

The result is in 2's complement form. The sign bit 1 is depicting it is negative number.

Case 3 Addition of two negative numbers.

$$\begin{array}{r}
 -7 \\
 -3 \\
 \hline
 -10
 \end{array}
 \quad \text{Sign bit} \rightarrow 1,1001$$

$$\begin{array}{r}
 1,1101 \\
 \hline
 11,0110
 \end{array}
 \quad \text{Ignore} \quad \text{The result is } 1,0110 \text{ which is } 2\text{'s complement representing } -10$$

The result generated above shows sign bit as 1, which represents negative result and it is correct. However, if the result generated is outside the allowed range of numbers, the sign bit will become incorrect, which indicates an error condition, e.g.,

$$\begin{array}{r}
 -8 \\
 -9 \\
 \hline
 -17
 \end{array}
 \quad \text{Sign bit} \rightarrow 1,1000$$

$$\begin{array}{r}
 1,0111 \\
 \hline
 10,1111 \rightarrow \text{Error}
 \end{array}$$

Sign bit should be 1 indicating negative, but it is 0 (zero) here and hence the result is incorrect. If the same addition is performed with 8 bits (1 bit reserved for sign bit) it'll show the correct result e.g.,

$$\begin{array}{r}
 -8 \\
 -9 \\
 \hline
 -17
 \end{array}
 \quad \text{Sign bit} \rightarrow 1,111\ 1000$$

$$\begin{array}{r}
 1,111\ 0111 \\
 \hline
 11,110\ 1111
 \end{array}
 \quad \text{Ignore} \quad \text{The result is } 1,110\ 1111 \text{ which is } 2\text{'s complement representing } -17.$$

Overflow

Some of the previous results were carefully chosen to the extent that the numbers all fell within the valid range for 2's complement representation with $N = 4$. What happens if the arithmetic result exceeds the valid range? An incorrect result is produced and the arithmetic system should have the capability of flagging an overflow. Consider the following two cases:

$$\begin{array}{r}
 (+6) \ 0110 \\
 (+7) \ 0111 \\
 \hline
 1101
 \end{array}
 \quad \neq (+13)_{2C}$$

$$\begin{array}{r}
 (-6) \ 1010 \\
 (-7) \ 1001 \\
 \hline
 1\ 0001
 \end{array}
 \quad \neq (-13)_{2C}$$

Incorrect result because of overflow.

The result contains an extra bit than capacity - OVERFLOW

We observe that overflow conditions are produced when the sign bits of the operands are the same and differ from the sign bit of the result. This can be incorporated into the hardware of an arithmetic processing unit so as to handle it properly by flagging overflow.

1.7.5 Binary Multiplication

Binary multiplication is also performed in the same manner as decimal multiplication is performed. To perform binary multiplication, you need to know the following four binary multiplication rules :

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

Now consider the following example, which multiplies 1011_2 with 101_2 .

$$\begin{array}{r} 1011 \\ \times 101 \\ \hline 1011 \\ 0000 \times \\ 1011 \times \times \\ \hline 110111 \end{array}$$

(Using the above rules)

You can see some more binary multiplication examples in the following lines.

Example 1.40 Multiply 1100_2 with 1010_2 .

Solution.

$$\begin{array}{r} 1100 \\ \times 1010 \\ \hline 0000 \\ 1100 \times \\ 0000 \times \times \\ 1100 \times \times \times \\ \hline 1111000 \end{array}$$

Example 1.41 Multiply 1100110_2 with 1000_2

Solution.

$$\begin{array}{r} 1100110 \\ \times 1000 \\ \hline 1100110000 \end{array}$$

Example 1.42 Multiply 1.01_2 with 10.1_2 .

Solution.

$$\begin{array}{r} 1.01 \\ \times 10.1 \\ \hline 101 \\ 000 \times \\ 101 \times \times \\ \hline 11.001 \end{array}$$

1.7.6 Binary Division

Binary division is again similar to that of decimal division. Similar to decimal number system, here also, division by zero is meaningless.

The division rules followed by binary division are :

$$0 + 1 = 0$$

$$1 + 1 = 1$$

$$\begin{array}{r} 101 \\ \hline 101 \end{array}$$

$$\begin{array}{r} 101 \\ \hline 101 \end{array}$$

$$\begin{array}{r} 101 \\ \hline 101 \\ 101 \\ \hline \times \end{array}$$

For instance, to divide 11001_2 with 101_2 , we may do it as follows :

The other binary division examples are given below.

Example 1.43. Divide the binary number 110110 with binary number 101 .

Solution.

1010	
$\overline{101 \quad }$	
101	
$\underline{\quad \quad \quad}$	
111	
$\underline{\quad \quad \quad}$	
101	
$\underline{\quad \quad \quad}$	
100	Remainder

Example 1.44. Divide 11101_2 with 1100_2 .

Solution.

10.011010101...	
$\overline{1100 \quad }$	
1100	
$\underline{\quad \quad \quad}$	
10100	
$\underline{\quad \quad \quad}$	
1100	
$\underline{\quad \quad \quad}$	
10000	
$\underline{\quad \quad \quad}$	
1100	
$\underline{\quad \quad \quad}$	
...	

1.8 Adding and Subtracting Octal Numbers

Like binary numbers, you can also add and subtract octal numbers. The method of adding/subtracting two octal numbers is similar to that of decimal addition and subtraction. That is, in addition of two digits, if the sum results in 2 digits, then a carry is added to the higher order digits. Similarly, a subtraction may involve borrowing from higher order digits. Following sections will make it clear.

1.8.1 Octal Addition

In order to add two octal numbers, we proceed by adding lower order digits individually, generating carries if any and moving to higher order digits. For instance, consider the following example :

$$3412)_8 + 1263)_8$$

$$\begin{array}{r} 3412 \\ = 1263 \\ \hline 4675 \end{array}$$

The above example is straight forward because no carries are generated. But when the sum of two individual digits results into a carry, it might be confusing as you need to add carries too. To avoid confusion, you can make use of following table.

Table 1.8 *Addition of two octal digits*

		Digit 2							
		0	1	2	3	4	5	6	7
Digit 1	0	0	1	2	3	4	5	6	7
	1	1	2	3	4	5	6	7	10
	2	2	3	4	5	6	7	10	11
	3	3	4	5	6	7	10	11	12
	4	4	5	6	7	10	11	12	13
	5	5	6	7	10	11	12	13	14
	6	6	7	10	11	12	13	14	15
	7	7	10	11	12	13	14	15	16

Now, if we have to add the following numbers

$$2764)_8 + 6435)_8$$

1 ← carry

Step 1	2 7 6 4	6 4 3 5	
			1

4 + 5 = 11 (from above table i.e., sum = 1, carry = 1)

Step 2.1 1 + 6 = 7 (carry + first digit at ten's place)

Step 2.2 7 + 3 = 12 (Adding lower digits ; from above table i.e., sum = 2 carry 1)

1 1 ← carries

∴	2 7 6 4	6 4 3 5	
			2 1

(7 + 3 = 12 from table 1.8 ; i.e., sum = 2, carry 1)

Step 3.1 $1+7=10$ (carry from last operation + first digit in hundred's place)

Step 3.2 $10+4$ (Now adding lower digits i.e., sum = 4, carry 1)

≈ 14

$$\begin{array}{r} 1 \ 1 \ 1 \\ 2 \ 7 \ 6 \ 4 \\ \vdots \\ 6 \ 4 \ 3 \ 5 \\ \hline 4 \ 2 \ 1 \end{array} \quad \leftarrow \text{carries}$$

Step 4.1 $1+2=3$ (carry + first digit at thousand's place)

Step 4.2 $3+6$ (Adding second digit)
 ≈ 11 (from table 1.8)

$$\begin{array}{r} 1 \ 1 \ 1 \\ 2 \ 7 \ 6 \ 4 \\ \vdots \\ 6 \ 4 \ 3 \ 5 \\ \hline 11 \ 4 \ 2 \ 1 \end{array} \quad \leftarrow \text{carries} \quad = \text{Result}$$

$$\text{Hence } 2764_8 + 6435_8 = 11421_8$$

Example 1.45. $1476_8 + 4454_8$

$$\begin{array}{r} 1 \ 1 \ 1 \\ 1 \ 4 \ 7 \ 6 \\ \vdots \\ 4 \ 4 \ 5 \ 4 \\ \hline \text{sum} = 6 \ 1 \ 5 \ 2 \end{array}$$

All digits are in octal.

Scratch pad

$$6+4=12)_8$$

$$1+7+5=15)_8$$

$$1+4+4=11)_8$$

$$1+1+4=6)_8$$

1.8.2 Octal Subtraction

You can subtract octal numbers similar to decimal numbers whereby you borrow from higher order digits if needed. But this procedure is confusing for octal subtraction. A simpler way of performing subtraction in octal numbers is by using 8's complement. So, we'll first learn to calculate 8's complement and then do the octal subtraction.

1.8.2A 8's Complement

The Eight's complement is found by first taking 7's complement and then adding 1 to it. The 7's complement is found by subtracting each digit of octal number from 7.

Example 1.46. To find 8's complement of $347)_8$.

Solution.

Step 1. Find 7's complement.

$$\text{i.e., } 777_8 - 347_8$$

$$\begin{array}{r} 7 \ 7 \ 7 \\ - 3 \ 4 \ 7 \\ \hline 4 \ 3 \ 0 \end{array}$$

Step 2. Add 1 to 7's complement

$$= 430 + 1 = 431$$

Hence 8's complement of $347)_8$ is 431.

Once you know how to find 8's complement, you can subtract two octal numbers.

Octal numbers are subtracted by the following methods :

- Convert the subtrahend to 8's complement.
- Add the resultant 8's complement to minuend.
- Drop the final carry, if any.

Now consider the following example.

Example 1.47. $745)_8 - 567)_8$

Solution.

Step 1. Determine 8's complement for $567)_8$

$$\begin{array}{r}
 7 \ 7 \ 7 \\
 - 5 \ 6 \ 7 \\
 \hline
 2 \ 1 \ 0 \quad (7\text{'s complement}) \\
 + 1 \\
 \hline
 2 \ 1 \ 1 \quad (8\text{'s complement})
 \end{array}$$

Step 2. Add 8's complement to minuend, i.e.,

$$\begin{array}{r}
 7 \ 4 \ 5 \\
 + 2 \ 1 \ 1 \\
 \hline
 \text{→} 1 \ 1 \ 5 \ 6
 \end{array}$$

Step 3. Drop this carry.

∴ Answer is $156)_8$

1.9 Adding and Subtracting Hexadecimal Numbers

Adding and subtracting hexadecimal is also similar to that of octal numbers. For adding, we start by adding individual digits from lower order to higher order, generating and adding carries if any. Let us learn to do this practically.

1.9.1 Adding two Hexadecimal Numbers

In order to add two hexadecimal numbers, we process by adding lower order individual digits to higher order digits e.g., consider the following example :

$$71A3)_{16} + 142B)_{16}$$

$$\begin{array}{r}
 7 \ 1 \ A \ 3 \\
 = 1 \ 4 \ 2 \ B \\
 \hline
 8 \ 5 \ C \ E
 \end{array}$$

The above example is straight forward because no carries are generated. But when any carries are generated then you need to add the carries too.

To add two hexadecimal digits, you may use following table.

Table 1.9 Addition of two Hexadecimal digits

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	19	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Now, if we have to add the following numbers :

$$\text{FACE}_{16} + \text{8973}_{16}$$

1 ← carry

$$\begin{array}{r} \text{Step 1} = \begin{array}{rrrr} \text{F} & \text{A} & \text{C} & \text{E} \\ 8 & 9 & 7 & 3 \end{array} \\ \hline \end{array}$$

1 (E + 3 = 11 from table 1.9 ; i.e., sum = 1, carry 1)

$$\text{Step 2.1} \quad 1 + C = D \quad (\text{carry} + \text{first digit at 2nd higher order place})$$

$$\text{Step 2.2} \quad D + 7 \quad (\text{Adding lower digit})$$

= 14 (from table 1.9 ; i.e., sum = 4, carry 1)

1 1 ← carries

$$\begin{array}{r} \therefore \begin{array}{rrrr} \text{F} & \text{A} & \text{C} & \text{E} \\ 8 & 9 & 7 & 3 \end{array} \\ \hline \text{4} & \text{1} \end{array}$$

Step 3.1 $1 + A = B$ (carry + first digit at 3rd higher order place)

Step 3.2 $B + 9$ (Adding lower digit)
 $= 14$ (from table 1.9 ; i.e., sum = 4, carry = 1)

$$\begin{array}{r} & 1 & 1 & 1 & \leftarrow \text{carries} \\ \therefore & F & A & C & E \\ & 8 & 9 & 7 & 3 \\ \hline & 4 & 4 & 1 \end{array}$$

Step 4.1 $1 + F = 10$

Step 4.2 $10 + 8 = 18$ (i.e., sum = 8, carry 1)

$$\begin{array}{r} & 1 & 1 & 1 & \leftarrow \text{carries} \\ \therefore & F & A & C & E \\ & 8 & 9 & 7 & 3 \\ \hline & 18 & 4 & 4 & 1 \end{array}$$

$$\text{Hence } \text{FACE}_{16} + 8973_{16} = 18441_{16}$$

Now consider one more example

Example 1.48. $1\text{C}\text{F}\text{D}_{16} + 1\text{A}\text{A}\text{0}_{16}$

Solution.

$$\begin{array}{r} 1 \quad 1 \\ 1 \text{ C} \text{ F} \text{ D} \\ 1 \text{ A} \text{ A} \text{ 0} \\ \hline 3 \text{ 7} \text{ 9} \text{ D}_{16} \end{array}$$

Scratch pad

$$\text{D} + 0 = \text{D}$$

$$\text{F} + \text{A} = 19 \quad (\text{sum} = 9, \text{carry } 1)$$

$$1 + \text{C} + \text{A} = 17 \quad (\text{sum} = 7, \text{carry } 1)$$

$$1 + 1 + 1 = 3$$

1.9.2 Hexadecimal Subtraction

You may subtract hexadecimal numbers similar to decimal numbers whereby you borrow from higher order digits, if needed. But this procedure may be confusing for hexadecimal subtraction.

A simpler way of performing subtraction in hexadecimal numbers is by using 16's complement. So, we'll first learn to calculate 16's complement and then do the hexadecimal subtraction.

1.9.2A 16's Complement

The 16's complement is found by first taking 15's complement and then adding 1 to it. The 15's complement is found by subtracting each digit of hexadecimal number from fifteen i.e., F_{16} .

Consider the following example,

Example 1.49. To find 16's complement of $C5A_{16}$.

Solution.

Step 1. Find 15's complement

$$\begin{array}{r} \text{i.e.,} & F & F & F \\ & C & 5 & A \\ \hline & 3 & A & 5 \end{array}$$

Step 2. Add 1 to 15's complement i.e.,

$$3A5 + 1 = 3A6$$

$$F - A = 5$$

$$F - 5 = A$$

$$F - C = 3$$

Hence $3A6$ is 16's complement of $C5A_{16}$.

Once you know how to find 16's complement, you can subtract two hexadecimal numbers. The method you need to follow to subtract two hexadecimal numbers is as follows:

- (i) Convert the subtrahend to 16's complement.
- (ii) Add the resultant 16's complement to minuend.
- (iii) Drop the final carry, if any.

Now consider the following example.

Example 1.50. Calculate $1E5_{16} - 177_{16}$.

Solution.

Step 1. Determine 15's complement for 177_{16} .

$$\begin{array}{r} F \quad F \quad F \\ - 1 \quad 7 \quad 7 \\ \hline E \quad 8 \quad 8 \end{array}$$

$$F-7=8$$

$$F-1=E$$

Step 2. Add 1 to 15's complement to get 16's complement.

$$E88 + 1 = E89$$

Step 3. Add 16's complement to minuend.

i.e.,

1	E	5	
+	E	8	9
			(16's complement of 177_{16})
→			<u>1 0 6 E</u>

Step 4. Drop this carry

Hence $1E5_{16} - 177_{16} = 06E_{16}$

1.10 Representing Characters in Memory

In addition to numerical data, a computer must be able to handle numerical information. In other words, a computer should recognize codes that represent letters of the alphabet, punctuation marks, and other special characters as well as numbers. These codes are called *alphanumeric codes*. A complete alphanumeric code would include the 26 lowercase letters, 26 uppercase letters, 10 numeric digits, 7 punctuation marks, and anywhere from 20 to 40 other characters, such as +, /, #, %, *, and so on. We can say that an alphanumeric code represents all of the various characters and functions that are found on a standard typewriter (or computer) keyboard.

ASCII Code

The most widely used alphanumeric code, the *American Standard Code for Information Interchange* (ASCII), is used in most microcomputers and minicomputers, and in many mainframes. The ASCII code (pronounced "askee") is a 7-bit code, and so it has $2^7 = 128$ possible code groups. This is more than enough to represent all of the standard keyboard characters as well as control functions such as the (RETURN) and (LINEFEED) functions.

Table 1.10 shows a partial listing of the ASCII code. In addition to the binary code group for each character, the table gives the octal and hexadecimal equivalents.

Table 1.10 Partial Listing of ASCII Code

Character	7-Bit ASCII	Octal	Hex	Character	7-Bit ASCII	Octal	Hex
A	100 0001	101	41	Y	101 1001	131	59
B	100 0010	102	42	Z	101 1010	132	5A
C	100 0011	103	43	0	011 0000	060	30
D	100 0100	104	44	1	011 0001	061	31
E	100 0101	105	45	2	011 0010	062	32
F	100 0110	106	46	3	011 0011	063	33
G	100 0111	107	47	4	011 0100	064	34
H	100 1000	110	48	5	011 0101	065	35
I	100 1001	111	49	6	011 0110	066	36
J	100 1010	112	4A	7	011 0111	067	37
K	100 1011	113	4B	8	011 1000	070	38
L	100 1100	114	4C	9	011 1001	071	39
M	100 1101	115	4D	blank	010 1000	040	20
N	100 1110	116	4E	.	010 1110	056	2E
O	100 1111	117	4F	(110 1000	050	28
P	101 0000	120	50	+	010 1011	053	2B
Q	101 0001	121	51	\$	010 0100	044	24
R	101 0010	122	52	*	010 1010	052	2A
S	101 0011	123	53)	010 1001	051	29
T	101 0100	124	54	-	010 1101	055	2D
U	101 0101	125	55	/	010 1111	057	2F
V	101 0110	126	56	,	010 1100	054	2C
W	101 0111	127	57	=	011 1101	075	3D
X	101 1000	130	58	(RETURN)	000 1101	015	0D
				(LINEFEED)	000 1010	012	0A

Example 1.51. The following is a message encoded in ASCII code. What is the message?

1001000 1000101 1001100 1010000

Solution. Convert each 7-bit code to its hex equivalent. The results are

48 45 4C 50

Now locate these hex values in Table 1.10 and determine the character represented by each. The results are

H E L P

The ASCII code is used for the transfer of alphanumeric information between a computer and input/output devices such as video terminals or printers. A computer also uses it internally to store the information that an operator types in at the computer's keyboard.

Example 1.52. An operator is typing in a BASIC program at the keyboard of a certain micro-computer. The computer converts each keystroke into its ASCII code and stores the code in memory. Determine the codes that will be entered into memory when the operator types in the following BASIC statement :

GOTO 25

Solution. Locate each character (including the space) in Table 1.10 and record its ASCII code.

G	1000111
O	1001111
T	1010100
O	1001111
(Space)	0100000
2	0110010
5	0110101

Apart from ASCII there are other systems that are also used to represent various symbols. Other than ASCII, EBCDIC and Unicode are also very popular.

EBCDIC stands for *Extended Binary Coded Decimal Interchange Code* and it is an eight-bit code capable of representing 256 symbols. EBCDIC is still used in IBM mainframe and mid-range systems but personal computers rarely used it.

Another popular character representation system is Unicode. This is new and evolving standard and getting more and more popular day by day. Unicode is 16-bit code and is capable of representing 65,536 different characters. The unicode is capable of representing Chinese, Korean, Japanese, Greek, and many other symbols along with all those represented by ASCII or EBCDIC.

Bits, Bytes and Words

As you've learnt that a *bit* represents single binary digit and a *byte* represents a group of 8-bit. A *word* represents amount of data i.e., group of bits, that a computer can represent and work upon at any given time. For example, if a computer is capable of processing 32-bits at a time, the word size is said to be 32-bits. The bigger the word size, the faster the computer can process a set of data.

1.11 ISCII

With the advent of computerization considerable work has been undertaken to facilitate the use of Indian languages on computers. These activities were generally limited to specific languages and were independent exercises of various organizations, thus, making data-interchange impossible. In such a scenario, it was important to have a common standard for coding Indian scripts. In 1991, the Bureau of Indian Standards adopted the *Indian Standard Code for Information Interchange (ISCII)*, the *ISCII standard* that was evolved by a standardization committee. This is an eight-bit code capable of coding 256 characters. ISCII code retains all ASCII characters and offers coding for Indian scripts also. Thus, it is also called *Indian Scripts Code for Information Interchange*.

All GIST products are based on ISCII. Also ISCII has been used by IBM for PC-DOS, Apple for ILK, and by several other companies that are developing products and solutions based on this representation. Also ISCII has been made mandatory for the data being collected by organizations like *The Election Commission*, and for projects as *Land Records Project* etc.

This standard does not only apply to the *Devanagari* script, but also to the *Gurmukhi*, *Gujarati*, *Oriya*, *Bengali*, *Assamese*, *Telugu*, *Kannada*, *Malayalam* and *Tamil* script. Because the structure of these scripts is so similar that a single coding can be applied to all of them, immediately providing *transliteration* between the scripts. Some of the ISCII versions for different scripts are being given below.

	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
240	ા	િ	િ	િ	ા	ા	િ	િ	િ	િ	િ	િ	િ	િ	િ	િ
260	ઓ	ઔ	ઔ	ક	ખ	ગ	ઘ	ડ	ચ	છ	જ	ઝ	બ	ટ	ર	ડ
300	ઢ	ણ	ત	ભ	દ	ધ	ન	X	પ	ફ	બ	ભ	મ	ય	X	ર
320	ર	લ	ળ	X	વ	શ	ષ	સ	હ	INV	ા	િ	િ	િ	િ	િ
340	X	િ	િ	િ	X	િ	િ	િ	િ	િ	િ	િ	િ	િ	િ	ATR
360	EXT	૦	૧	૨	૩	૪	૫	૬	૭	૮	૯	X	X	X	X	X

Devnagari ISCII script © ISCII

	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
240	ા	િ	િ	િ	ા	ા	િ	િ	િ	િ	િ	િ	િ	િ	િ	િ
260	ও	ঔ	X	ক	খ	গ	ঘ	ঙ	চ	ছ	জ	ঝ	ব	ট	ঠ	ড
300	ঢ	ণ	ত	ভ	দ	ধ	ন	X	প	ফ	ব	ভ	ম	য	ৰ	ৰ
320	X	ল	X	X	X	X	X	X	স	হ	INV	া	ি	ি	ি	ি
340	X	০	১	X	X	X	X	X	৮	৯	।	X	X	X	X	ATR
360	EXT	૦	૧	૨	૩	૪	૫	૬	૭	૮	૯	X	X	X	X	X

Bengali ISCII script © ISCII

	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
240	ા	િ	િ	િ	ા	ા	િ	િ	િ	િ	િ	િ	િ	િ	િ	િ
260	ઓ	ઔ	X	ક	ખ	ગ	ઘ	ડ	ચ	છ	જ	ઝ	બ	ટ	ર	ડ
300	ঢ	ণ	ত	ভ	দ	ধ	ন	X	প	ফ	ব	ভ	ম	য	ৰ	ৰ
320	X	ল	X	X	X	X	X	X	স	হ	INV	া	ি	ি	ি	ি
340	X	০	১	X	X	X	X	X	৮	৯	।	X	X	X	X	ATR
360	EXT	૦	૧	૨	૩	૪	૫	૬	૭	૮	૯	X	X	X	X	X

Gujarati ISCII script © ISCII

1.12 Unicode

After discussing ISCII, let us now talk about Unicode, the new universal coding standard being adopted all newer platforms. It has been developed by *Unicode Consortium* that was formed in 1991. As officially stated,

*Unicode provides a unique number for every character,
no matter what the platform,
no matter what the program,
no matter what the language.*

Fundamentally, computers just deal with numbers. They store letters and other characters by assigning a number for each one. Before Unicode was invented, there were hundreds of different encoding systems for assigning these numbers. No single encoding could contain enough characters: for example, the European Union alone requires several different encodings to cover all its languages. Even for a single language like English, no single encoding was adequate for all the letters, punctuation, and technical symbols in common use.

These encoding systems also conflict with one another. That is, two encodings can use the same number for two different characters, or use different numbers for the same character. Any given computer (especially servers) needs to support many different encodings; yet whenever data is passed between different encodings or platforms, that data always runs the risk of corruption.

Significance

- Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. The Unicode Standard has been adopted by such industry leaders as Apple, HP, IBM, JustSystem, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many others. Unicode is required by modern standards such as XML, Java, ECMAScript (JavaScript), LDAP, CORBA 3.0, WML, etc., and is the official way to implement ISO/IEC 10646. Unicode is becoming very popular and is supported in many operating systems, all modern browsers, and many other products. The emergence of the Unicode Standard, and the availability of tools supporting it, are among the most significant recent global software technology trends.
- ◆ Incorporating Unicode into client-server or multi-tiered applications and websites offers significant cost savings over the use of legacy character sets.
- ◆ Unicode enables a single software product or a single website to be targeted across multiple platforms, languages and countries without re-engineering.
- ◆ Unicode allows data to be transported through many different systems without corruption.

Characters Represented

Unicode version 3.0 represented 49,194 characters, whereas Unicode version 3.1 has added many more characters, making the character count to 94,140.

Indian Languages on Unicode

The Unicode Standard has incorporated Indian scripts under the group named *Asian Scripts* (Chapter 9, Unicode Standard 3.0). The Indian scripts included are *Devnagari, Bengali, Gurumukhi, Gujarati, Oriya, Tamil, Telugu, Kannada* and *Malayalam*. The Indian language block of Unicode Standard is based on ISCII-1988.

We are giving below *Devanagari* and *Bengali* scripts as given in chapter 9 of Unicode 3.0.

	090	091	092	093	094	095	096	097
0	ऐ	ठ	र	ी	ॐ	ऋ	०	
1	ॐ	आ	ड	र	ु	ঁ	ৱ	
2	ঁ	ও	ঢ	ল	ূ	ু	৩	
3	০:	আ	ণ	঳	০	ৈ	৩	
4	ঔ	আু	ত	঳	ঁ	ী	—	
5	অ	ক	থ	ব	ু	ু	॥	
6	আ	খ	দ	শা	ু	ু	০	
7	ই	গ	ধ	প	ু	ু	১	
8	ঁ	ঁ	ন	স	ু	ু	২	
9	ঁ	ঁ	ন	হ	ু	ু	৩	
A	ঁ	ঁ	প		ু	ু	৪	
B	ঁ	ঁ	ফ		ু	ু	৫	
C	ঁ	জ	ব	ঁ	ু	ঁ	৬	
D	ঁ	ঁ	ভ	ঁ	ু	ঁ	৭	?
E	ঁ	ঁ	ম	ঁ	ু	ঁ		
F	ঁ	ট	য	ি	ু	ঁ		

	098	099	09A	09B	09C	09D	09E	09F
0	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ
1	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ
2	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ
3	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ
4	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ	ঁ
5	অ	ক	থ					
6	আ	খ	দ	শ			০	৤
7	ই	গ	ধ	ষ	ু	ু	১	
8	ঁ	ঁ	ঁ	ঁ	ু	ু	২	
9	ঁ	ঁ	ঁ	ঁ	ু	ু	৩	
A	ঁ	ঁ	ঁ	ঁ	ু	ু	৪	
B	ঁ	ঁ	ঁ	ঁ	ু	ু	৫	
C	ঁ	ঁ	ঁ	ঁ	ু	ু	৬	
D	ঁ	ঁ	ঁ	ঁ	ু	ু	৭	
E	ঁ	ঁ	ঁ	ঁ	ু	ু	৮	
F	ঁ	ঁ	ঁ	ঁ	ু	ু	৯	

Devanagari Script

Copyright © Unicode Inc.

Bengali Script

Let Us Revise

- ❖ The decimal number system (base -10) is composed of 10 unique symbols : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- ❖ The binary number system (base -2) is composed of 2 unique symbols : 0 and 1.
- ❖ The octal number system (base-8) is composed of 8 unique symbols : 0, 1, 2, 3, 4, 5, 6, 7.
- ❖ The hexadecimal (hex) number system (base -16) is composed of 16 symbols : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

- ❖ The decimal, binary, octal and hexadecimal systems are positional-value systems wherein each component digit carries certain weight depending upon its position.
- ❖ In a number of any number system, the rightmost digit carries the least weight, known as least significant digit, and the left most digit carries the largest weight, known as most significant digit.
- ❖ Integers can be represented in binary form in three ways : sign and magnitude representation, one's complement representation, and two's complement representation.
- ❖ In sign and magnitude representation, the MSB -stores 0 for positive numbers and 1 for negative numbers.
- ❖ 1's complement of a binary number is calculated by replacing every 0 with 1 and every 1 with 0.
- ❖ 2's complement of a binary number is calculated by adding 1 to its 1's complement.
- ❖ Real numbers are represented by their exponents and mantissa.
- ❖ Characters are represented in memory by alpha-numeric codes. One such code is ASCII (American Standard Code for Information Interchange).
- ❖ ASCII, EBCDIC and Unicode are popular character representation systems.

Solved Problems

1. Convert the following decimal numbers to binary : (i) 13 (ii) 106 (iii) 84

Solution.

(i)

2	13
2	6
2	3
2	1
	0

Remainders

1
0
1
1

$$13_{10} = 1101_2$$

(ii)

2	106
2	53
2	26
2	13
2	6
2	3
2	1
	0

Remainders

0
1
0
1
0
1
1

$$106_{10} = 1101010_2$$

(iii)

2	84
2	42
2	21
2	10
2	5
2	2
2	1
	0

Remainders

0
0
1
0
1
0
1

$$84_{10} = 1010100_2$$

2. Convert the following binary numbers to decimal : (i) 10010 (ii) 101010 (iii) 1010100.011

Solution. (i) $10010_2 =$

$$\begin{aligned} &= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 16 + 0 + 0 + 2 + 0 = 18 = 18_{10} \end{aligned}$$

(ii) 101010_2

$$\begin{aligned}
 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
 &= 32 + 0 + 8 + 0 + 2 + 0 \\
 &= 42_{10}
 \end{aligned}$$

(iii) 1010100.011_2

$$\begin{aligned}
 &= 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\
 &= 64 + 0 + 16 + 0 + 4 + 0 + 0 + 0 + 0.25 + 0.125 \\
 &= 84.325_{10}
 \end{aligned}$$

3. Convert 22.25_{10} to binary.

Solution.

		Remainder	Fractional Part	
2	22	0	0.25	$0.25 \times 2 = 0.50$
2	11	1	0.50	$0.50 \times 2 = 1.00$
2	5	1		
2	2	0		
2	1	1		
	0			

$= 1011001_2$

4. Convert 11011110101110_2 to hexadecimal.

Solution.

Group in fours

11,0111,1010,1110

Convert each number

3 7 A E

Thus, the solution is 37AE.

5. Convert $4A8C_{16}$ to binary.

Solution.

Given :

4 A 8 C

Convert each digit :

0100 1010 1000 1100

Thus, the solution is $100,1010,1000,1100_2$ 6. Convert FACE₁₆ to binary.

Solution.

Given :

F A C E

Convert each digit :

1111 1010 1100 1110

Thus, the solution is 1111,1010,1100,1110₂.7. Convert $2C9_{16}$ to decimal.

Solution.

$$2C9_{16} = 2 \times 16^2 + 12 \times 16^1 + 9 \times 16^0$$

$$= 512 + 192 + 9 = 713_{10}$$

Note that a C represents a decimal 12.

8. Convert EB4A₁₆ to decimal.

Solution.

$$\begin{aligned} \text{EB4A}_{16} &= 14 \times 16^3 + 11 \times 16^2 + 4 \times 16^1 + 10 \times 16^0 \\ &= 57,344 + 2816 + 64 + 10 = 60,234_{10} \end{aligned}$$

9. Convert 423₁₀ to hexadecimal.

Solution.

Successive Division	Remainders	HexNotation
16) 423		
16) 26	7	7
16) 1	10	A
0	1	1

Reading the remainders up from the bottom, the result is 1A7₁₆.

10. Convert 72905₁₀ to hexadecimal.

Solution.

Successive Division	Remainders	Hex Notation
16) 72905		
16) 4556	9	9
16) 284	12	C
16) 17	12	C
16) 1	1	1
0	1	1

Reading the remainders up from the bottom, the result is 11,CC9₁₆.

11. Convert 1111011110101₂ to octal.

Solution.

Divide into groups of three.

11,111,011,110,101

Express each group in decimal.

3 7 3 6 5

Therefore, 1111011110101₂ = 37,365₈.

12. Convert 101110100011000111₂ to octal.

Solution.

Divide into groups.

1,011,110,100,011,000,111

Express groups in decimal.

1 3 6 4 3 0 7

Therefore, 101110100011000111₂.

= 1,364,307₈

13. Convert 3674₈ to binary.

Solution.

Copy the octal number.

3 6 7 4

Convert each to binary.

011 110 111 100

Therefore, 3674₈ = 11,110,111,100₂

14. What negative value does 1001 1011 represent?

Solution. The two's complement of 1001 1011 is 0110 0101. This represents a 101₁₀.

Therefore, 1001 1011₂ = -101₁₀.

15. Convert $B2F_{16}$ to octal.

Solution. It's easiest to first convert hex to binary, then to octal.

$$\begin{aligned} B2F_{16} &= 1011 \quad 0010 \quad 1111 && \{\text{convert to binary}\} \\ &= 101 \quad 100 \quad 111 && \{\text{group bits by 3s}\} \\ &= 5 \quad 4 \quad 5 \quad 7_8 && \{\text{convert to octal}\} \end{aligned}$$

16. Add the binary numbers 110101 and 101111.

Solution.

1	1	0	1	0	1
1	0	1	1	1	1
<hr/>					
1	1	0	0	1	0

17. Add the binary numbers 10110 and 1101.

Solution.

1	0	1	1	0
1	1	0	1	
<hr/>				
1	0	0	0	1

18. Subtract 101111 from 110101.

Solution.

1	1	0	1	0	1
1	0	1	1	1	1
<hr/>					
0	0	0	1	1	0

19. Subtract 1101 from 10110.

Solution.

1	0	1	1	0
0	1	1	0	1
<hr/>				
1	0	0	1	

20. Multiply 10110 with 1101.

Solution.

1	0	1	1	0
	×	1	1	0
<hr/>				
1	0	1	1	0
0	0	0	0	×
1	0	1	1	0
1	0	1	1	0
<hr/>				
1	0	0	1	1

21. Multiply 111 with 101.

Solution.

1	1	1
	×	1
<hr/>		
1	1	1
0	0	0
1	1	1
<hr/>		
1	0	0

22. Divide 11110 by 110.

Solution.

$$\begin{array}{r} 101 \\ 110 \overline{)11110} \\ 110 \\ \hline 110 \\ 110 \\ \hline 0 \end{array}$$

23. What are the other systems apart from ASCII that represent various symbols?

Solution. Apart from ASCII, other systems that are used to represent various symbols are EBCDIC and Unicode. EBCDIC stands for Extended Binary Coded Decimal Interchange code and it is an eight-bit code capable of representing 256 symbols.

Unicode is 16-bit code and is capable of representing 65,536 different characters.

24. What are the limitations of fixed point numbers?

Solution. The first drawback of this scheme is the need of the user to remember and keep track of the decimal point location. The second drawback of this scheme is that the range of numbers which can be represented, using this scheme is limited to 999.999.

25. Describe the different formats with reference to binary floating-point numbers.

Solution. Precision is the degree to which the correctness of a quantity is expressed. Single-precision, double-precision and extended-precision binary floating-point numbers have the same basic formats except for the number of bits. Single-precision floating-point numbers have 32 bits, double-precision numbers have 64 bits, and extended precision numbers have 80 bits.

Glossary

ASCII American Standard Code for Information Interchange. It is a 7-bit code to represent characters.

Binary number system A number system with 2 unique symbols i.e., base-2 system.

Decimal Number System A number system with 10 unique symbols i.e., base-10 system.

Hexadecimal Number System A number system with 16 unique symbols i.e., base-16 system.

Number system An organized way of representing numbers.

Octal Number System A number system with 8 unique symbols i.e., base-8 system.

EBCDIC Extended Binary Coded Decimal Interchange Code. It is an 8-bit code used to represent various symbols.

A Assignments

TYPE A : VERY SHORT ANSWER QUESTIONS

- What are the bases of decimal, octal, binary and hexadecimal systems?
- What is the common property of decimal, octal, binary, and hexadecimal?

3. Complete the sequence of following binary numbers :
100, 101, 110, _____, _____, _____, _____
4. Complete the sequence of following octal numbers :
525, 526, 527, _____, _____, _____
5. Complete the sequence of following hexadecimal numbers 17, 18, 19, _____, _____
6. When and why are octal and hexadecimal numbers preferred over binary numbers ?
7. Encode "SANNU" and "KUSHAGRA" into ASCII codeform.
8. ASCII code is _____ bit code.
9. What is the use of ASCII code ?
10. What is the full form of ASCII and EBCDIC ?
11. What does a word represent ?
12. What is ISCII ? How many bits are used by ISCII ?
13. What is Unicode ? How many characters are represented by it ?
14. Does unicode support Indian Scripts ? If yes, then name them.

TYPE B : SHORT ANSWER QUESTIONS

1. Convert the following binary numbers to decimal :
 - (a) 1010
 - (b) 111010
 - (c) 101011111
2. Convert the following binary numbers to decimal :
 - (a) 1100
 - (b) 10010101
 - (c) 11011100
3. Convert the following decimal numbers to binary :
 - (a) 23
 - (b) 100
 - (c) 145
4. Convert the following decimal numbers to binary :
 - (a) 19
 - (b) 121
 - (c) 161
5. Convert the following hexadecimal numbers to binary :
 - (a) A6
 - (b) A07
 - (c) 7AB4
6. Convert the following hexadecimal numbers to binary :
 - (a) BE
 - (b) BC9
 - (c) 9BC8
7. Convert the following binary numbers to hexadecimal :
 - (a) 10011011101
 - (b) 1111011101011011
 - (c) 11010111010111
8. Convert the following binary numbers to hexadecimal :
 - (a) 101011011011
 - (b) 10110111011011
 - (c) 1111101110101111
9. Convert the following hexadecimal numbers to decimal :
 - (a) A6
 - (b) A13B
10. Convert the following hexadecimal numbers to decimal :
 - (a) E9
 - (b) 7 CA3
11. Convert the following decimal numbers to hexadecimal :
 - (a) 132
 - (b) 2352
12. Convert the following decimal numbers to hexadecimal :
 - (a) 206
 - (b) 3619
13. Convert the following hexadecimal numbers to octal :
 - (a) 38 AC
 - (b) 7FD6
 - (c) ABCD
14. Convert the following octal numbers to binary :
 - (a) 123
 - (b) 3527
15. Convert the following octal numbers to binary :
 - (a) 7642
 - (b) 7015
 - (c) 3576

16. Convert the following binary numbers to octal :
 (a) 111010 (b) 110110101 (c) 1101100001
17. Convert the following binary numbers to octal :
 (a) 11001 (b) 10011101 (c) 111010111
18. Find the eight-bit two's complement form of the following decimal numbers :
 (a) -10 (b) -52 (c) -123
19. Find the eight-bit two's complement form of the following decimal numbers :
 (a) -14 (b) -49 (c) -99
20. Convert the following decimal numbers to 10-bit binary :
 (a) 37.31 (b) 6.215 (c) 33.333
21. Convert the following decimal numbers to 10-bit binary :
 (a) 27.2 (b) 3.21 (c) 63.362
22. Convert the following binary numbers to decimal :
 (a) 101.01 (b) 101.1101 (c) 11101.1111
23. Convert the following binary numbers to decimal :
 (a) 110.11 (b) 110.1001 (c) 10101.0101
24. Encode the following message in ASCII code using the hex representation "COST = \$72."
25. The following ASCII-coded message is stored in successive memory locations in a computer :
 1010011 10101000 1001111, 101000.
 What is the message ?
26. Convert each octal number to its decimal equivalent :
 (a) 743 (b) 36 (c) 3777 (d) 257 (e) 1204
27. Convert each of the following decimal numbers to octal and perform addition of resultant numbers :
 (a) with (b) ; (c) with (d) and (e) with (a) :
 (a) 59 (b) 372 (c) 919 (d) 65,535 (e) 255
28. Convert these hex values to decimal :
 (a) 92 (b) 1A6 (c) 37FD (d) 2C0 (e) 7FF
29. Convert these decimal values to hex and perform addition of resultant hex numbers as :
 (a) with (b) ; (c) with (d) and (d) with (e)
 (a) 75 (b) 314 (c) 2048 (d) 25,619 (e) 4095
30. Perform subtraction of following octal numbers
 (a) 743 - 36 (b) 3777 - 1204 (c) 1204 - 743
31. Perform subtraction of following hexadecimal numbers
 (a) 1A6 - 92 (b) 37FD - 7FF (c) 2C0 - 1A6
32. Add the following binary numbers :
 (i) 10110111 and 1100101
 (ii) 110101 and 101111
 (iii) 110111.110 and 11011101.010
 (iv) 1110.110 and 11010.011
33. Perform Q. 32 using 1's complement and 2's complement arithmetic where consider the left most bit as sign bit.

34. Subtract the following :
(i) 100101100 from 1110101010
(ii) 10001.0011 from 11011.011
(iii) 101 from 110011
35. Perform Q. 34 using 1's complement and 2's complement arithmetic where consider the left most bit as sign bit.
36. Multiply the following :
(i) 10011 by 1101 (ii) 110.101 by 1011.001
37. Divide the following binary numbers :
(i) 1011 by 011 (ii) 110111 by 1011
38. Subtract the following using 1's and 2's complement :
(i) 10101 - 10001 (ii) 11011 - 1110 (iii) 100100 - 100011 (iv) 11011.110 - 101.001
39. What do you think is the significance of Unicode ?

TYPE C : LONG ANSWER QUESTIONS

1. How are data represented in memory ? Discuss the representation schemes for numerical and alphanumerical data.
2. How are data represented in memory ? Discuss the representation schemes for numerical and alphanumerical data.
3. Discuss different ways to add and subtract using 1's complement and 2's complement.

Propositional Logic & Hardware

In This Chapter

2.1 Introduction

Studying logic is important because it provides us a way to support our claims to truth. It is tempting to say that logic arguments establish the truth of their conclusions. As a field of study, we can define logic in many ways. We may say that logic is all about *arguments*. It is a formal method of reasoning. In the context of logic, an argument is not a quarrel or dispute, but an example of reasoning where a statement offers support, justification, ground, reason or evidence for another statement.

As earlier said that *logic is a formal method for reasoning*. Logic can be symbolically represented in many ways. One such way of doing so is *propositional logic*. This chapter is dedicated to the study of *propositional logic, digital logic and hardware*. Let us begin with the discussion of propositional logic.

2.2 Propositional Logic

The propositional logic represents logic through *propositions* and *logical connectives*. We may define *proposition* as an elementary atomic sentence that may take either *true* value or *false* value but may not take any other value.

2.1 Introduction

2.2 Propositional Logic

2.3 Basic Logic Gates

2.4 More about Logic Gates

2.5 Applications of Logic Gates

Consider the following examples :

It is raining.	[It is a proposition as it may either be true or false]
Australia have won ICC World Cup 2007.	[It is also a proposition as it is true]
India is a continent.	[It is a proposition as it is false]
What did you eat ?	[It is not a proposition as it does not result in true or false]
How are you ?	[Not a proposition for the similar reason as above]

Propositions are also called *sentence* or *statements*. After this introduction, let us now talk about terms and symbols used in propositional logic.

A **Proposition** is an elementary atomic sentence that may either be true or false but may take no other value.

2.2.1 Terms and Symbols

A *simple proposition* is one that does not contain any other proposition as a part. We will use the lower-case letters, p, q, r, \dots , as symbols for simple statements or propositions.

A *compound proposition* is one with two or more simple propositions as parts or what we will call *components*. A component of a compound is any whole proposition that is part of a larger proposition ; components may themselves be compounds.

For example, following are compound propositions :

It is raining and wind is blowing.

Take it or leave it.

If you work hard then you will be rewarded.

An **operator** (or *connective*) joins simple propositions into compounds, and joins compounds into larger compounds. We will use the symbols, $+, ., \Rightarrow$, and \Leftrightarrow to designate the sentential connectives. They are called *sentential* connectives because they join *sentence* (or what we are calling *statements* or *propositions*). The symbol, \sim , is the only operator that is not a connective ; it affects single statements only, and does not join statements into compounds.

The symbols for statements and for operators comprise our notation or symbolic language. Parentheses serve as punctuation.

Different types of **connectives** (or operators) used in propositional logic are as given below :

1. **Disjunctive** (Also called OR). Represented by symbols $+$ or \vee . Disjunction means one of the two arguments is *true* or both e.g., $p + q$ (or $p \vee q$) means p OR q . Its meaning is either p is true, or q is true, or both.
2. **Conjunctive** (Also called AND). Represented by symbols $.$ or $\&$ or \wedge . Conjunction means both arguments are true e.g., $p . q$ (or $p \& q$) means p AND q . Its meaning is both p and q are true.
3. **Conditional** (Also called If.. Then or Implication). Represented by symbols \Rightarrow or \rightarrow or \supset . Implication means if one argument is true then other argument is true e.g., $p \Rightarrow q$ (or $p \supset q$ or $p \rightarrow q$) means If p then q . Its meaning is if p is true, then q is true.
4. **Bi-conditional** (Also called If and only If or Equivalence). Represented by symbols \Leftrightarrow or \equiv .

Equivalence (or bi-conditional) means either both arguments are true or both are false, e.g., $p \Leftrightarrow q$ (or $p \equiv q$) means if and only if p is true then q is true. Its meaning is p and q are either both true or both false.

5. Negation (Also called NOT). Not a connective actually, just an operator. Represented by \sim or ' or $\bar{}$ (bar). It is an operator that affects a single statement only and does not join two or more statements e.g., $\sim p$ (or p' or \bar{p}) means NOT p . Its meaning is p is false.

We can summarize the above discussion as follows :

Simple statements

p	" p is true"	assertion
$\sim p$	" p is false"	negation

Compounds and connectives

$p + q$	"either p is true, or q is true, or both"	disjunction
$p \cdot q$	"both p and q are true"	conjunction
$p \Rightarrow q$	"if p is true, then q is true"	implication
$p \Leftrightarrow q$	" p and q are either both true or both false"	equivalence

Implication statements ($p \Rightarrow q$) are sometimes called *conditionals*, equivalence statements ($p \Leftrightarrow q$) are sometimes called *biconditionals*.

Well-formed Formulae

As mentioned earlier, propositions are also called sentences or statements. Another term *formulae* or *well-formed formulae* also refer to the same. That is, we may also call *well-formed-formula (wff)* to refer to a proposition.

2.2.2 Truth values and Wff

Every simple or compound proposition may take an either *true value* or *false value*. These *true* (also denoted by 1) or *false* (also denoted by 0) are also called *truth values*. We may define *truth value* as *truth* or *falsity* of a proposition.

The *truth value* of a statement is its truth or falsity. All meaningful statements have truth values, whether they are **simple** or **compound, asserted or negated**. That is, p is either true or false, $\sim p$ is either true or false, $p + q$ is either true or false, and so on.

A compound statement is *truth-functional* if its truth value as a whole can be figured out solely on the basis of the truth values of its parts or components. A connective is truth-functional if it makes only compounds that are truth-functional. For example, if we knew the truth values of p and of q , then we could figure out the truth value of the compound, $p + q$. Therefore the compound, $p + q$, is a truth-functional compound and disjunction is a truth-functional connective.

All four of the connectives we are studying (disjunction, conjunction, implication, and equivalence) are truth-functional. Negation is a truth-functional operator. With these four connectives and negation we can express *all* the truth-functional relations among statements. A truth table helps us express it.

Truth value is defined as truth or falsity of a proposition.

A **Truth Table** is a complete list of possible truth values of a proposition.

Let us now learn to make truth tables for all the connectives, we have learnt so far.

(i) Negation (NOT)

The NOT operator works on single proposition, thus, it is also called *unary connective* sometimes. If p denotes a proposition, then its negation will be denoted by $\sim p$ or p' or \bar{p} . If p is 0 (false), then $\sim p$ is 1 (true) and if p is 1 (true) then $\sim p$ is 0 (false). The truth table for this operation is shown as follows :

Table 2.1 Truth table for Negation (NOT)

p	\bar{p}
0	1
1	0

Also note that

NOT (NOT p) results into p itself i.e.,

$$\bar{\bar{p}} = p$$

$$\text{or } (p')' = p$$

$$\text{or } \sim(\sim p) = p.$$

(ii) Disjunction (OR)

The OR connective works with more than one proposition. The compound $p + q$ has two (2) component propositions (p and q), each of which can be true or false. So there are four (2^2) possible combinations. The disjunction of p with q (denoted as $p + q$ or $p \vee q$) will be true whenever p is true or q is true or both are true. Consider the truth table given below :

Table 2.2 Truth table for Disjunction (OR)

p	q	$p + q$
0	0	0
0	1	1
1	0	1
1	1	1

Note If a compound has n distinct components, there will be 2^n rows in its truth table.

(iii) Conjunction (AND)

The AND connective also works with more than one proposition. The compound $p \cdot q$ (or $p \& q$) will be *true* whenever both p and q are true.

Table 2.3 Truth table for Conjunction (AND)

p	q	$p \cdot q$
0	0	0
0	1	0
1	0	0
1	1	1

(iv) Implication (If.. Then / Conditional)

In the conditional $p \Rightarrow q$, the first proposition (the if-clause) p here, is called the *antecedent* and the second proposition (then clause) q here, is called the *consequent*. In more complex conditionals, the *antecedent* and the *consequent* could themselves be *compound propositions*. The conditional $p \Rightarrow q$ will be *false* when p is *true* and q is *false*. For all other input combinations, it will be *true*.

Table 2.4 Truth table for If.. Then

p	q	$p \Rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

The conditional $p \Rightarrow q$ may be expressed as follows :

$$p \Rightarrow q = p' + q$$

(v) Equivalence (If and only If / Bi-conditional)

A bi-conditional results into *false* when one of its component propositions is *true* and the other is *false*. That is, $p \Leftrightarrow q$ will be 0 (false) when p is 0 and q is 1 Or p is 1 and q is 0. For all other inputs, $p \Leftrightarrow q$ is 1.

Table 2.5 Truth table for If and only if

p	q	$p \Leftrightarrow q$
0	0	1
0	1	0
1	0	0
1	1	1

The bi-conditional $p \Leftrightarrow q$ may also be expressed as :

$$p \Leftrightarrow q = pq + p' \cdot q'$$

Some Related Terms

Contingencies

The propositions that have some combination of 1's and 0's in their truth table column, are called *contingencies*.

Tautologies

The propositions having nothing but 1's in their truth table column, are called *tautologies*.

Contradictions

The propositions having nothing but 0's in their truth table column, are called *contradictions*.

Consistent Statements

Two statements are *consistent* if and only if their conjunction is *not a contradiction*.

Converse

The converse of a conditional proposition is determined by interchanging the antecedent and consequent of given conditional. It results into a new conditional.
e.g., Converse of $p \Rightarrow q$ is $q \Rightarrow p$.

That is, if

p : It is raining.

q : Sky is not clear.

then, $p \Rightarrow q$ = If it is raining then sky is not clear.

Its converse will be new conditional as given below :

$q \Rightarrow p$ = If sky is not clear then it is raining.

Inverse

The inverse of a conditional proposition is another conditional having negated antecedent and consequent. That is, the inverse of $p \Rightarrow q$ is $\neg p \Rightarrow \neg q$. e.g., if

p : It is raining.

q : Sky is not clear.

then, $p \Rightarrow q$ = If it is raining then sky is not clear.

Its inverse will be a new conditional as given below :

$\neg p \Rightarrow \neg q$ = If it is not raining then sky is clear.

Contrapositive The contrapositive of a conditional is formed by creating another conditional that takes its antecedent as negated consequent of earlier conditional and consequent as negated antecedent of earlier conditional. That is, contrapositive of $p \Rightarrow q$ is $\neg q \Rightarrow \neg p$ or $\neg q \Rightarrow \neg p$ or $\neg p \Rightarrow \neg q$

2.2.3 Some Equivalence Propositional Laws

Two sentences are equivalent if they have the same truth value under every interpretation i.e., both the sentences possess the same truth set.

In the following lines, we are giving some equivalence laws used in propositional logic. We are giving them without proofs, since their proofs are beyond the scope of this book.

Table 2.6 Some Equivalence Laws

1. $0 + p = p$	<i>Properties of 0</i>	7. $p + q = q + p$	<i>Commutative law</i>
$0 \cdot p = 0$		$p \cdot q = q \cdot p$	
2. $1 + p = 1$	<i>Properties of 1</i>	8. $(p + q) + r = p + (q + r)$	<i>Associative law</i>
$1 \cdot p = p$		$(p \cdot q) \cdot r = p \cdot (q \cdot r)$	
3. $p + pq = p$	<i>Absorption law</i>	9. $p \cdot (q + r) = (p \cdot q) + (p \cdot r)$	<i>Distributive law</i>
$p + (p + q) = p$		$p + (q \cdot r) = (p + q) \cdot (p + r)$	
4. $\bar{p} = p$	<i>Involution</i>	$p + \bar{p} \cdot q = p + q$	
5. $p + \bar{p} = p$	<i>Idempotence law</i>	10. $\begin{array}{l} p + q = \bar{p} \cdot \bar{q} \\ p \cdot q = \bar{p} + \bar{q} \end{array}$	<i>De Morgan's law</i>
$p \cdot \bar{p} = 0$			
6. $p + \bar{p} = 1$	<i>Complementarity law</i>	11. $p \Rightarrow q = \bar{p} + q$	<i>Conditional elimination</i>
$p \cdot \bar{p} = 0$		12. $p \Leftrightarrow q = (p \Rightarrow q) \cdot (q \Rightarrow p)$	<i>Bi-conditional elimi.</i>

Let us now have a look at some examples.

Example 2.1. Construct a truth table for the expression $(A \cdot (A + B))$. What single term is the expression equivalent to?

Solution.

A	B	$A + B$	$(A \cdot (A + B))$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Looking at the table, we find that columns $(A \cdot (A + B))$ and A are identical. That is, they possess the same *truth set*. Hence the given expression $(A \cdot (A + B))$ is equivalent to A.

Example 2.2. Using truth table, prove that $p \Rightarrow q$ is equivalent to $\sim q \Rightarrow \sim p$.

Solution.

p	q	$\sim q$	$\sim p$	$p \Rightarrow q$	$\sim q \Rightarrow \sim p$
0	0	1	1	1	1
0	1	0	1	1	1
1	0	1	0	0	0
1	1	0	0	1	1

From the above truth table it is obvious that columns $p \Rightarrow q$ and $\sim q \Rightarrow \sim p$ are identical i.e., possessing same truth set (1, 1, 0, 1). Hence it is proved that

$$p \Rightarrow q = \sim q \Rightarrow \sim p.$$

This rule is also called *transposition*.

Example 2.3. Prove that $p \Rightarrow q = \bar{p} + q$.

Solution.

p	q	\bar{p}	$p \Rightarrow q$	$\bar{p} + q$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	0	1	1

From the above truth table, we find that columns $p \Rightarrow q$ and $\bar{p} + q$ are possessing same *truth set* (1, 1, 0, 1). Hence proved that

$$p \Rightarrow q = \bar{p} + q.$$

Example 2.4. Prove that $p \Leftrightarrow q = q \Leftrightarrow p$.

Solution.

p	q	$p \Leftrightarrow q$	$q \Leftrightarrow p$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	1	1

From the above truth table, we find that both propositions $p \Leftrightarrow q$ and $q \Leftrightarrow p$, possess the same truth set (1, 0, 0, 1). Hence proved that

$$p \Leftrightarrow q = q \Leftrightarrow p.$$

Example 2.5. Prove that $p \Leftrightarrow q = (p \Rightarrow q) \cdot (q \Rightarrow p)$.

Solution.

p	q	$p \Leftrightarrow q$	$p \Rightarrow q$	$q \Rightarrow p$	$(p \Rightarrow q) \cdot (q \Rightarrow p)$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	1	1	1	1

Since the columns $p \Leftrightarrow q$ and $(p \Rightarrow q) \cdot (q \Rightarrow p)$ are identical, it is proved that

$$p \Leftrightarrow q = (p \Rightarrow q) \cdot (q \Rightarrow p)$$

Example 2.6. Consider some simple propositions given below :

A : It is raining. B : Wind is blowing. C : I am not driving.

From these, create the following compound propositions :

- (i) $A \vee B$ (ii) $\sim B$ (iii) $\sim B \cdot C$ (iv) $A \cdot \sim C$ (v) $A + B \cdot C$.

Solution.

- (i) $A \vee B$: It is raining OR wind is blowing.
- (ii) $\sim B$: Wind is NOT blowing.
- (iii) $\sim A \cdot C$: It is NOT raining AND I am not driving.
- (iv) $A \cdot \sim C$: It is raining AND I am driving.
- (v) $A + B \cdot C$: It is raining OR wind is blowing AND I am not driving.

Example 2.7. Prove that $X + 1$ is a tautology.

Solution.

X	1	$X + 1$
0	1	1
1	1	1

Since the column $X + 1$ has all trues (1's) in its column, it is a tautology.

Example 2.8. Prove that $X + X'$ is a tautology and $X \cdot X'$ is a contradiction.

Solution.

X	X'	$X + X'$	$X \cdot X'$
0	1	1	0
1	0	1	0

$X + X'$ has all 1's in its truth set, hence it is a tautology.

$X \cdot X'$ has all 0's in its truth set, hence it is a contradiction.

2.2.4 Drawing Conclusions – Syllogism

While studying logic, many a times conclusions are drawn from given two or more logic statements. This process, rather logical process of drawing conclusions from given logic statements, is called *syllogism*. The given statements or propositions are called *premises*.

To draw conclusions, we may use any of the *two* methods available for it :

- ◆ Truth Table Method
- ◆ Algebraic Method

1. Truth Table Method

In this method, a truth table (TT) is drawn for all the *given premises* and the *conclusion to be drawn*. Then a conditional is prepared having the *antecedent as conjunction of all given premises* and *consequent as the conclusion to be drawn*. If this conditional results into a tautology (all true's i.e., 1's) then the given conclusion is established. For this consider the example given below :

Example 2.9. From premises p and $p \Rightarrow q$, infer q .

Solution. Given premises are :

$$(P1) : p$$

$$(P2) : p \Rightarrow q$$

Conclusion to be drawn $C : q$

As per the TT method, we have to prepare a truth table for given premises p , $p \Rightarrow q$, the conjunction of these given premises i.e., $p . (p \Rightarrow q)$ and the conclusion to be drawn i.e., q .

Also there should be a column for

$$P1 . P2 \Rightarrow C \text{ i.e., } [p . (p \Rightarrow q)] \Rightarrow q$$

Our truth table will look like :

p	q	$p \Rightarrow q$	$p . (p \Rightarrow q)$	$[p . (p \Rightarrow q)] \Rightarrow q$
0	0	1	0	1
0	1	1	0	1
1	0	0	0	1
1	1	1	1	1

From the above truth table, it is clear that the conditional having *antecedent* as the conjunction of premises $[p . (p + q)]$ and *consequent* as the conclusion (q), is a tautology. Hence it is established that from given premises p and $p \Rightarrow q$ conclusion drawn is q .

That is

$$\frac{p \\ p \Rightarrow q}{q}$$

The logical process of drawing conclusions from given propositions is called *syllogism*. The propositions used to draw conclusion are called *premises*.

Example 2.10. From $p \Rightarrow q$ and $q \Rightarrow r$, infer $p \Rightarrow r$.

Solution. Given premises are :

$$(P1) : p \Rightarrow q$$

$$(P2) : q \Rightarrow r$$

Conclusion (C) to be drawn : $p \Rightarrow r$ as per 'TT' method.

We shall prepare a truth table having column for

$$P1 \text{ (i.e., } p \Rightarrow q\text{), } P2 \text{ (i.e., } q \Rightarrow r\text{), } P1 \cdot P2 \text{ i.e., } (p \Rightarrow q) \cdot (q \Rightarrow r),$$

$$C \text{ (i.e., } p \Rightarrow r\text{) and } P1 \cdot P2 \Rightarrow C \text{ i.e., } [(p \Rightarrow q) \cdot (q \Rightarrow r)] \Rightarrow (p \Rightarrow r)$$

p	q	r	$p \Rightarrow q$	$q \Rightarrow r$	$(p \Rightarrow q) \cdot (q \Rightarrow r)$	$p \Rightarrow r$	$[(p \Rightarrow q) \cdot (q \Rightarrow r)] \Rightarrow (p \Rightarrow r)$
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	1	0	1	0	0	1	1
0	1	1	1	1	1	1	1
1	0	0	0	1	0	0	1
1	0	1	0	1	0	1	1
1	1	0	1	0	0	0	1
1	1	1	1	1	1	1	1

From the table, we derive that

$P1 \cdot P2 \Rightarrow C$ is a tautology i.e., having all 1's in its truth set. Hence concluded that

$$p \Rightarrow q$$

$$q \Rightarrow r$$

$$\frac{}{p \Rightarrow r}$$

(ii) Algebraic Method

In this method, to draw a conclusion from given premises, **conditional elimination** is carried out. That is, in place of a conditional $p \Rightarrow q$ its equivalent $\bar{p} + q (\sim p + q)$ is substituted and then it is checked whether the conditional having antecedent as the conjunction-of-all premises and consequent as the conclusion-to-be-drawn, is a tautology or not. To understand this, consider the example given below.

Example 2.11. From p and $p \Rightarrow q$, infer q .

Solution. Given premises are

$$(P1) : p$$

$$(P2) : p \Rightarrow q$$

Conclusion (C) to be drawn : q .

Let us compute $P1 \cdot P2 \Rightarrow C$

$$\text{i.e., } [p \cdot (p \Rightarrow q)] \Rightarrow q$$

Carrying out conditional elimination i.e., substituting $p \Rightarrow q$ with $\bar{p} + q$, we get

$$= [p \cdot (\bar{p} + q)] \Rightarrow q$$

$$= (p \cdot \bar{p} + p \cdot q) \Rightarrow q$$

$$= 0 + (p \cdot q) \Rightarrow q$$

$$(\because p \bar{p} = 0)$$

Carrying out conditional elimination once again, we get

$$\begin{aligned}
 & (\overline{p \cdot q}) + q \\
 &= \overline{p} + \overline{q} + q \quad (\because \overline{pq} = \overline{p} + \overline{q}, \text{ Rule 10, table 2.6}) \\
 &= \overline{p} + 1 \quad (\because \overline{q} + q = 1, \text{ Rule 6, table 2.6}) \\
 &= \overline{p} + 1 \quad (\because \overline{p} + 1 = \overline{p}, \text{ Rule 2, table 2.6}) \\
 &= 1
 \end{aligned}$$

Hence the result is established.

Example 2.12. From $p \Rightarrow q$ and $q \Rightarrow r$, infer $p \Rightarrow r$.

Solution. Given premises are : $p \Rightarrow q, q \Rightarrow r$

and conclusion to be drawn is $p \Rightarrow r$.

Thus we have to establish that

$$[(p \Rightarrow q) \cdot (q \Rightarrow r)] \Rightarrow (p \Rightarrow r)$$

Carrying out conditional elimination, we get

$$[(\overline{p} + q) \cdot (\overline{q} + r)] \Rightarrow (\overline{p} + r)$$

Carrying out conditional elimination once again, we get

$$\begin{aligned}
 & = (\overline{\overline{p} + q}) \cdot (\overline{\overline{q} + r}) + (\overline{p} + r) \\
 &= (\overline{\overline{p}} + \overline{q}) + (\overline{\overline{q}} + \overline{r}) + (\overline{p} + r) \quad (\text{De Morgan's Law}) \\
 &= (\overline{\overline{p}} \cdot \overline{q}) + (\overline{\overline{q}} \cdot \overline{r}) + (\overline{p} + r) \quad (\text{De Morgan's Law}) \\
 &= p \cdot \overline{q} + q \cdot \overline{r} + \overline{p} + r \\
 &= \overline{p} + p \cdot \overline{q} + r + q \cdot \overline{r} \quad [\text{Commutative law table 2.6}] \\
 &= (\overline{p} + p) \cdot (\overline{p} + \overline{q}) + (r + q)(r + \overline{r}) \quad [\text{Distributive law } p + qr = (p + q)(p + r)] \\
 &= 1 \cdot (\overline{p} + \overline{q}) + (r + q) \cdot 1. \quad [\because \overline{p} + p = r + \overline{r} = 1] \\
 &= \overline{p} + \overline{q} + r + q = \overline{p} + r + \overline{q} + q \quad [\because \overline{q} + q = 1] \\
 &= p + r + 1 \quad [\because r + 1 = 1] \\
 &= p + 1 \quad [\because p + 1 = 1]
 \end{aligned}$$

Hence the result is established.

The inference rules established in above examples are also known as *Modus ponens* and *Chain rule*. That is, **Modus Ponens** is

$$\frac{p}{p \Rightarrow q}$$

Chain rule is

$$p \Rightarrow q$$

$$\frac{q \Rightarrow r}{p \Rightarrow r}$$

Let Us Revise

- ❖ Logic is a formal method for reasoning.
- ❖ A proposition is an elementary atomic sentence that may either be true or false but may take no other value.
- ❖ A simple proposition is one that does not contain any other proposition as a part.
- ❖ A compound proposition is one with two or more simple propositions as parts.
- ❖ A connective joins simple propositions into compounds and joins compounds into larger compounds.
- ❖ Different types of connectives are disjunctive, conjunctive, conditional, bi-conditional and negation.
- ❖ The negation is a unary connective.
- ❖ Well-formed formula (wff) refers to sentences or propositions or statements.
- ❖ Truth value is defined as truth or falsity of a proposition.
- ❖ A truth table is a complete list of possible truth values of a proposition.
- ❖ The logical process of drawing conclusion from propositions is called syllogism.
- ❖ The propositions used to draw conclusion are called premises.
- ❖ Two inference rules are Modus Ponens and Chain rule.
- ❖ Modus ponens is.

$$\frac{p}{p \Rightarrow q}$$

$$\frac{p \Rightarrow q}{q}$$

- ❖ Chain rule is

$$\frac{p \Rightarrow q}{\frac{q \Rightarrow r}{p \Rightarrow r}}$$

2.3 Basic Logic Gates

System of logic was constructed long long ago by Aristotle. Long ago Aristotle constructed a complete system of formal logic and wrote six famous works on the subject, contributing greatly to the organization of man's reasoning. For centuries afterward, mathematicians kept on trying to solve these logic problems using conventional algebra but only George Boole could manipulate these symbols successfully to arrive at a solution with his own mathematical system of logic. Boole's revolutionary paper '*An Investigation of the laws of the thought*' was published in 1854 which led to the development of new system, the *algebra of logic*, 'BOOLEAN ALGEBRA'.

Boole's work remained confined to papers only until 1938 when Claude E. Shannon wrote a paper titled '*A Symbolic Analysis of Relay Switching Circuits*'. In this paper he applied Boolean Algebra to solve relay logic problems. As logic problems are binary decisions and boolean algebra effectively deals with these binary values. Thus it is also called '*Switching Algebra*'.

After Shannon applied boolean algebra in telephone switching circuits, engineers realized that boolean algebra could be applied to computer electronics as well.

In the computers, these boolean operations are performed by logic gates.

What is a Logic Gate ?

Gates are digital (two-state) circuits because the input and output signals are either low voltage (denotes 0) or high voltage (denotes 1). Gates are often called *logic circuits* because they can be analyzed with boolean algebra.

There are *three* basic logic gates and *four* others :

1. Inverter (NOT gate)
2. OR gate
3. AND gate

A gate is simply an electronic circuit which operates on one or more signals to produce an output signal.

2.3.1 Inverter (NOT Gate)

An inverter is also called a NOT gate because the output is not the same as the input. The output is sometimes called the *complement* (opposite) of the input.

Following tables summarize the operation.

An Inverter is a gate with only one input signal and one output signal ; the output state is always the opposite of the input state.

Table 2.7 Truth Table for NOT gate

X	\bar{X}
Low	High
High	Low

Table 2.8 Alternative truth table for NOT gate

X	\bar{X}
0	1
1	0

A low input i.e., 0 produces high output i.e., 1, and vice versa. The symbol for inverter is given in adjacent Fig. 2.1.

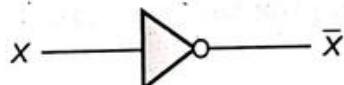


Figure 2.1 NOT gate symbol

2.3.2 OR Gate

If all inputs are 0 then output is also 0. If one or more inputs are 1, the output is 1.

An OR gate can have as many inputs as desired. No matter how many inputs are there, the action of OR gate is the same : one or more 1 (high) inputs produce output as 1. Following tables show OR action :

The OR gate has two or more input signals but only one output signal. If any of the input signals is 1 (high), the output signal is 1 (high).

Table 2.9 Truth Table for two input OR gate

X	Y	F
0	0	0
0	1	1
1	0	1
1	1	1

$$F = X + Y$$

Table 2.10 Truth Table for three input OR gate

X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F = X + Y + Z$$

The symbol for OR gate is given below :

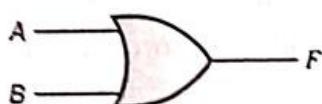
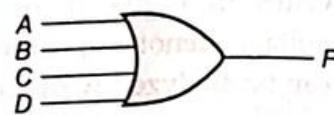


Figure 2.2 (a) Two input OR gate



(b) Three input OR gate



(c) Four input OR gate.

2.3.3 AND gate

If any of the inputs is 0, the output is 0. To obtain output as 1, all inputs must be 1.

An AND gate can have as many inputs as desired. Following tables illustrate AND action.

The AND Gate can have two or more than two input signals and produce one output signal. When all the inputs are 1 i.e., high then the output is 1 otherwise output is 0.

Table 2.11 Two input AND gate

X	Y	F
0	0	0
0	1	0
1	0	0
1	1	1

Here, $F = X \cdot Y$

Table 2.12 Three input AND gate

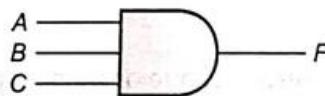
X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Here,
 $F = X \cdot Y \cdot Z$

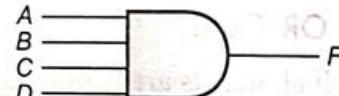
The symbol for AND is



Figure 2.3 (a) 2-input AND gate



(b) 3-input AND gate



(c) 4-input AND gate

2.4 More About Logic Gates

We have covered *three* basic logic gates NOT, OR, AND so far. But there are some more logic gates also which are derived from three basic gates (*i.e.*, AND, OR and NOT). These gates are more popular than NOT, OR and AND and are widely used in industry. This section introduces NOR, NAND, XOR, XNOR gates.

2.4.1 NOR Gate

If either of the two input is 1 (*high*), the output will be 0 (*low*). NOR gate is nothing but inverted OR gate.

The NOR gate can have as many inputs as desired. No matter how many inputs are there, the action of NOR gate is the same *i.e.*, all 0 (*low*) inputs produce output as 1.

Following truth Tables (2.13 and 2.14) illustrate NOR action.

The NOR gate has two or more input signals but only one output signal. If all the inputs are 0 (*low*), then the output signal is 1 (*high*)

Table 2.13 2-input NOR gate

X	Y	F
0	0	1
0	1	0
1	0	0
1	1	0



Table 2.14 3-input NOR gate

X	Y	Z	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

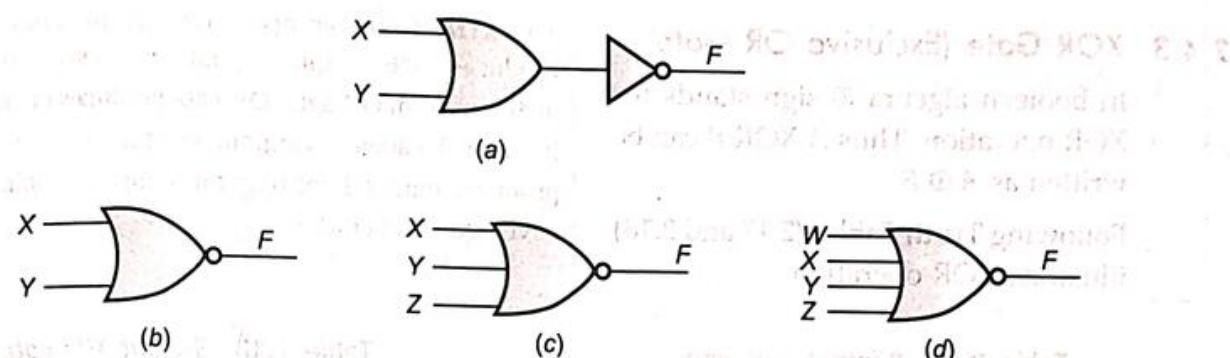
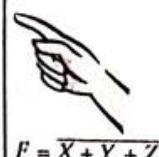


Figure 2.4 (a) Logical meaning of NOR gate (b) 2 input NOR gate (c) 3 input NOR gate (d) 4 input NOR gate

2.4.2 NAND Gate

NAND gate is inverted AND gate. Thus, for all 1 (high) inputs, it produces 0 (low) output, otherwise for any other input combination, it produces a 1 (high) output. NAND gate can also have as many inputs as desired.

The NAND gate has two or more input signals but only one output signal. If all of the inputs are 1 (high), then the output produced is 0 (low).

NAND action is illustrated in following Truth Tables (2.15 and 2.16).

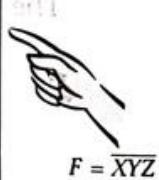
Table 2.15 2-input NAND gate's table

X	Y	F
0	0	1
0	1	1
1	0	1
1	1	0

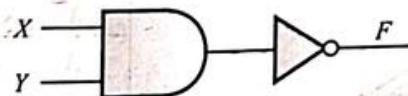


Table 2.16 3-input NAND gate's table

X	Y	Z	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



The logical meaning of NAND gate can be shown as follows :



The symbols of 2, 3, 4 input NAND gates are given below :

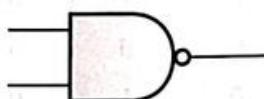
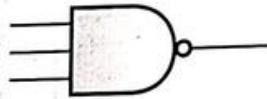


Figure 2.5

(a) 2-input NAND gate



(b) 3-input NAND gate



(c) 4-input NAND gate.

2.4.3 XOR Gate (Exclusive OR Gate)

In boolean algebra \oplus sign stands for XOR operation. Thus $A \text{ XOR } B$ can be written as $A \oplus B$.

Following Truth Tables (2.17 and 2.18) illustrate XOR operation.

The XOR Gate can also have two or more inputs but produces one output signal. Exclusive-OR gate is different from OR gate. OR gate produces output 1 for any input combination having one or more 1's, but XOR gate produces output 1 for only those input combinations that have odd number of 1's.

Table 2.17 2-input XOR gate.

No. of 1's even/odd	X	Y	F
Even	0	0	0
Odd	0	1	1
Odd	1	0	1
Even	1	1	0

Table 2.18 3-input XOR gate.

No. of 1's	X	Y	Z	F
Even	0	0	0	0
Odd	0	0	1	1
Odd	0	1	0	1
Even	0	1	1	0
Odd	1	0	0	1
Even	1	0	1	0
Even	1	1	0	0
Odd	1	1	1	1

Note

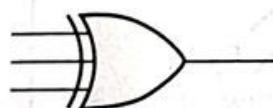
Remember odd number of 1's produce output 1 in XOR operation.

The symbols of XOR gates are given below :

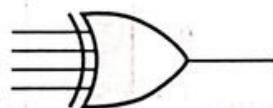


Figure 2.6

(a) 2-input XOR gate



(b) 3-input XOR gate



(c) 4-input XOR gate.

XOR addition can be summarized as follows :

$$0 \oplus 0 = 0; \quad 0 \oplus 1 = 1; \quad 1 \oplus 0 = 1; \quad 1 \oplus 1 = 0$$

2.4.4 XNOR Gate (Exclusive NOR gate)

Following Tables (2.19 and 2.20) illustrate XNOR action.

The XNOR Gate is logically equivalent to an inverted XOR i.e., XOR gate followed by a NOT gate (inventor). Thus XNOR produces 1 (high) output when the input combination has even number of 1's.

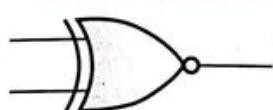
Table 2.19 2 input XNOR gate

No. of 1's even/odd	X	Y	F
Even	0	0	1
Odd	0	1	0
Odd	1	0	0
Even	1	1	1

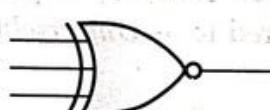
Table 2.20 3-input XNOR gate

No. of 1's	X	Y	Z	F
Even	0	0	0	1
Odd	0	0	1	0
Odd	0	1	0	0
Even	0	1	1	1
Odd	1	0	0	0
Even	1	0	1	1
Even	1	1	0	1
Odd	1	1	1	0

Following are the XNOR gate symbols :



(a) 2-input XNOR gate



(b) 3-input XNOR gate



(c) 4-input XNOR gate.

The bubble (small circle), on the outputs of NAND, NOR, XNOR gates represents complementation.

Now that you are familiar with logic gates, you can use them in designing logic circuits.

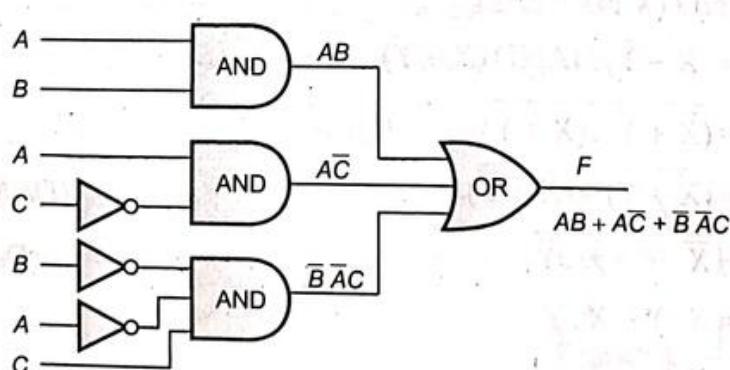
Example 2.13. Design a circuit to realize the following : $F(a, b, c) = AB + A\bar{C} + \bar{B}\bar{A}\bar{C}$

Solution. The given boolean expression can also be written as follows :

$$F(a, b, c) = A \cdot B + A \cdot \bar{C} + \bar{B} \cdot \bar{A} \cdot C$$

or $F(a, b, c) = (A \text{ AND } B) \text{ OR } (A \text{ AND } (\text{NOT } C)) \text{ OR } ((\text{NOT } B) \text{ AND } (\text{NOT } A) \text{ AND } C)$

Now these logical operators can easily be implemented in form of logic gates. Thus circuit diagram for above expression will be as follows.



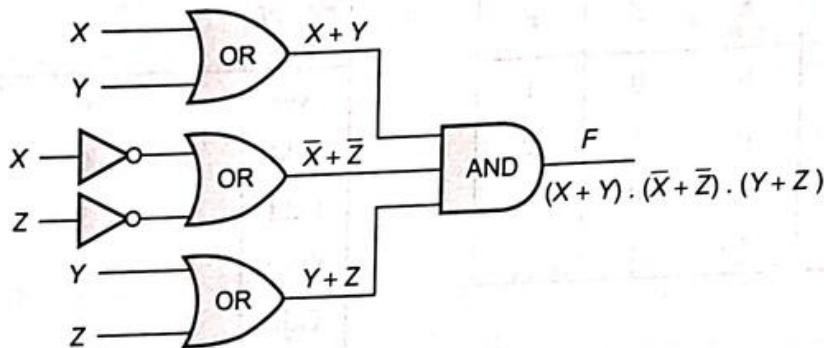
Example 2.14. Draw the diagram of digital circuit for the function

$$F(X, Y, Z) = (X + Y) \cdot (\bar{X} + \bar{Z}) \cdot (Y + Z)$$

Solution. Above expression can also be written as

$$F(X, Y, Z) = (X \text{ OR } Y) \text{ AND } ((\text{NOT } X) \text{ OR } (\text{NOT } Z)) \text{ AND } (Y \text{ OR } Z)$$

Thus circuit diagram will be



2.4.5 NAND to NAND and NOR to NOR Design

We can design circuits using AND, OR, NOT gates as we have done so far, but NAND and NOR gates are more popular as these are less expensive and easier to design. And also other switching functions (AND, OR) can easily be implemented using NAND/NOR gates. Thus NAND, NOR gates are also referred to as *Universal Gates*.

NAND-to-NAND Logic

AND and OR operations from NAND gates are shown below :

AND Operation

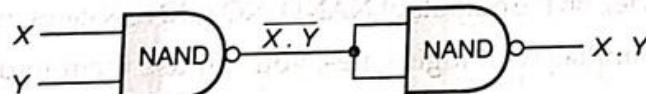


Figure 2.8 AND operation using NAND-NAND logic.

AND operation using NAND is

$$X \cdot Y = (X \text{ NAND } Y) \text{ NAND } (X \text{ NAND } Y)$$

Proof. $X \text{ NAND } Y = \overline{X \cdot Y}$

$$= \overline{X} + \overline{Y}$$

(De Morgan's Second Theorem)

$$(X \text{ NAND } Y) \text{ NAND } (X \text{ NAND } Y)$$

$$= (\overline{X} + \overline{Y}) \text{ NAND } (\overline{X} + \overline{Y})$$

$$= \overline{(\overline{X} + \overline{Y}) \cdot (\overline{X} + \overline{Y})}$$

$$= \overline{(\overline{X} \cdot \overline{Y}) + (\overline{X} \cdot \overline{Y})}$$

$$= \overline{\overline{X}} \cdot \overline{\overline{Y}} + \overline{\overline{X}} \cdot \overline{\overline{Y}}$$

$$= \overline{X} \cdot \overline{Y} + \overline{X} \cdot \overline{Y}$$

$$= X \cdot Y + X \cdot Y$$

(De Morgan's Second Theorem)

(De Morgan's First Theorem)

$$(\overline{\overline{X}} = X)$$

$$(X + X = X)$$

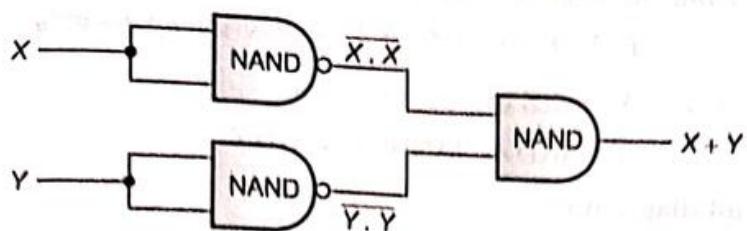
OR Operation

Figure 2.9 OR operation using NAND to NAND logic.

OR operation using NAND is

$$X + Y = (X \text{ NAND } X) \text{ NAND } (Y \text{ NAND } Y)$$

Proof. $X \text{ NAND } X = \overline{X} \cdot \overline{X}$

$$= \overline{X} + \overline{X}$$

$$= \overline{X}$$

(De Morgan's Second Theorem)
 $(X + X = X)$

Similarly, $Y \text{ NAND } Y = \overline{Y}$

Therefore, $(X \text{ NAND } X) \text{ NAND } (Y \text{ NAND } Y)$

$$= \overline{X} \text{ NAND } \overline{Y}$$

$$= \overline{\overline{X}} \cdot \overline{\overline{Y}}$$

$$= \overline{\overline{X}} + \overline{\overline{Y}}$$

$$= X + Y$$

(De Morgan's Second Theorem)
 $(\overline{\overline{X}} = X, \overline{\overline{Y}} = Y)$

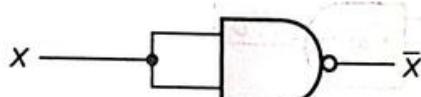
Not Operation

Figure 2.10 NOT operation using NAND logic.

NOT operation using NAND gates is

$$\text{NOT } X = X \text{ NAND } X$$

Proof. $X \text{ NAND } X = \overline{X} \cdot \overline{X} = \overline{X}$

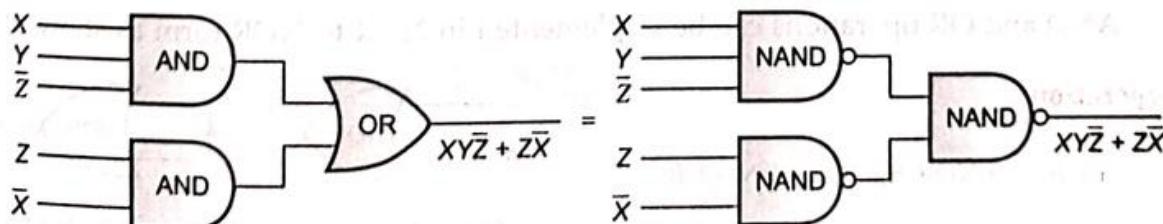
$(\because X \cdot X = X)$

NAND-to-NAND logic is best suited for boolean expression in *Sum-of-Products* form.

Design rule for NAND-TO-NAND logic Network (only for two-level-circuits)

- Derive simplified sum-of-products expression.
- Draw a circuit diagram using AND, OR gates
- Just replace AND and OR gates with NAND gates

For example, $XY\bar{Z} + Z\bar{X}$ can be drawn as follows :



(a) AND-to-OR implementation.

(b) NAND-to-NAND implementation.

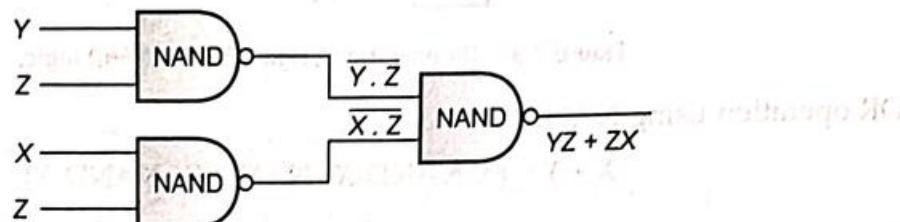
Example 2.15. Draw the diagram of a digital circuit for the function

$$F(X, Y, Z) = YZ + XZ \text{ using NAND gates only.}$$

Solution. $F(X, Y, Z) = YZ + XZ$ can be written as

$$= (Y \text{ NAND } Z) \text{ NAND } (X \text{ NAND } Z)$$

Thus logic circuit diagram is



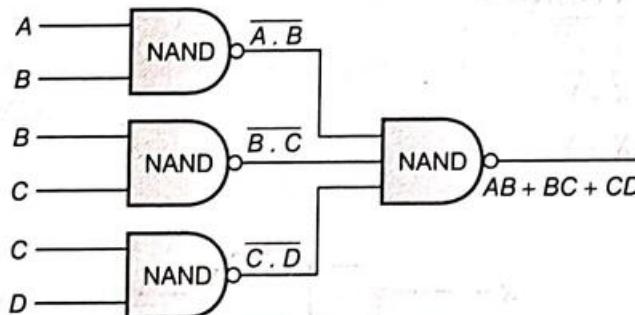
Example 2.16. Draw the diagram of digital circuit for

$$F(a, b, c) = AB + BC + CD \text{ using NAND-to-NAND logic.}$$

Solution. $F(a, b, c) = AB + BC + CD$

$$= (A \text{ NAND } B) \text{ NAND } (B \text{ NAND } C) \text{ NAND } (C \text{ NAND } D)$$

Thus logic circuit is



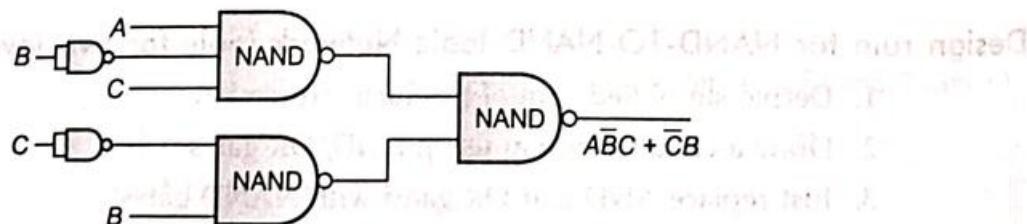
Example 2.17. Draw the circuit diagram for

$$F = A\bar{B}C + \bar{C}B \text{ using NAND-to-NAND logic only.}$$

Solution. $F = A\bar{B}C + \bar{C}B$

$$= ((A) \text{ NAND } (\text{NOT } B) \text{ NAND } (C)) \text{ NAND } ((\text{NOT } C) \text{ NAND } B)$$

Thus logic circuit is



NOR-to-NOR Logic

AND and OR operations can be implemented in NOR-to-NOR form as shown below :

OR operation

$$A + B = (A \text{ NOR } B) \text{ NOR } (A \text{ NOR } B)$$

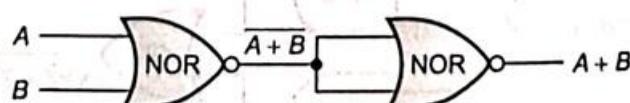


Figure 2.12 OR operation using NOR to NOR logic.

AND operation

$$A \cdot B = (A \text{ NOR } A) \text{ NOR } (B \text{ NOR } B)$$

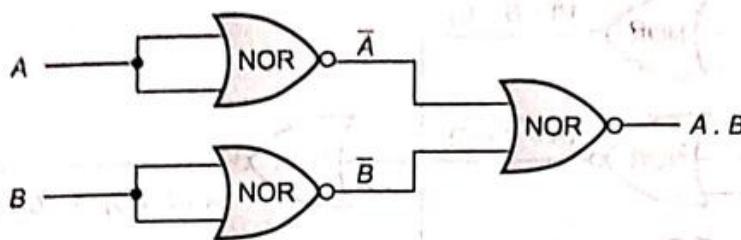


Figure 2.13 AND operation using NOR to NOR logic.

NOT Operation

$$\text{NOT } A = A \text{ NOR } A$$

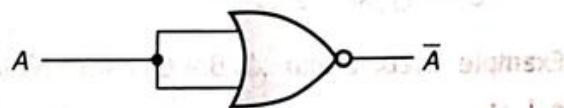


Figure 2.14 NOT operation using NOR Logic

NOR-to-NOR logic is best suited for boolean expression in Product-of-Sums form.

Design rule for NOR-to-NOR logic network (only for 2-level-circuits)

- Derive a simplified Product-of-Sums form of the expression.
- Draw a circuit diagram using OR, AND gates
- Finally substitute NOR gates for OR, NOT and AND gates

For example, $(X + Y)(Y + Z)(Z + X)$ can be implemented as follows :

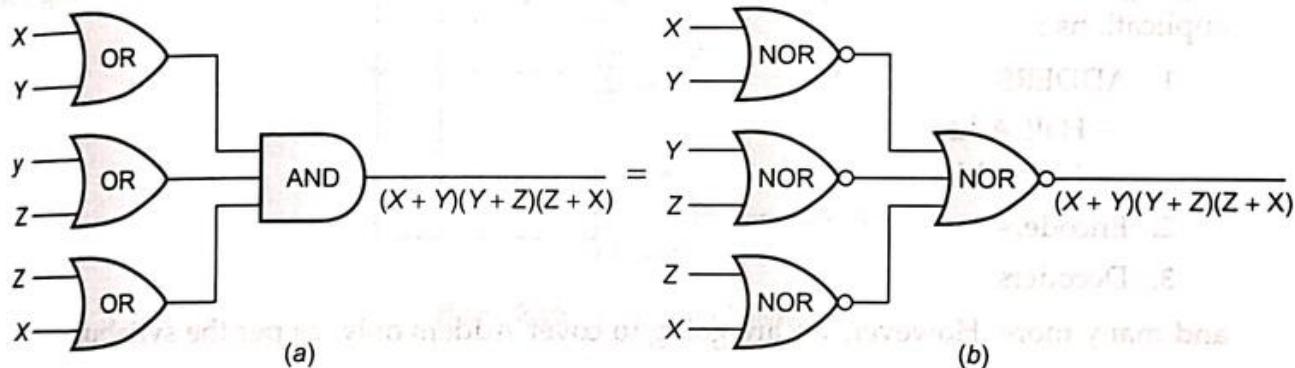
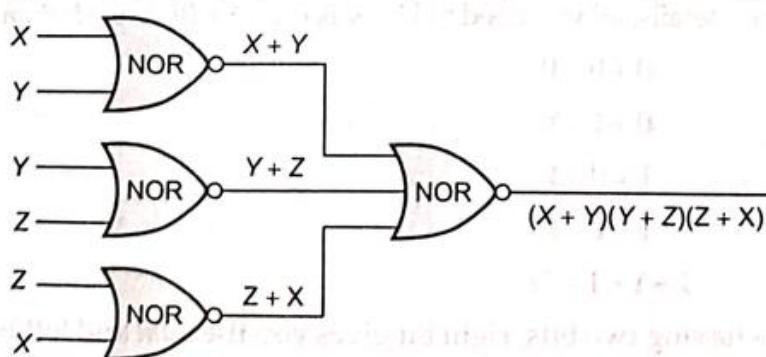


Figure 2.15 (a) OR-to-AND implementation (b) NOR-to-NOR circuit.

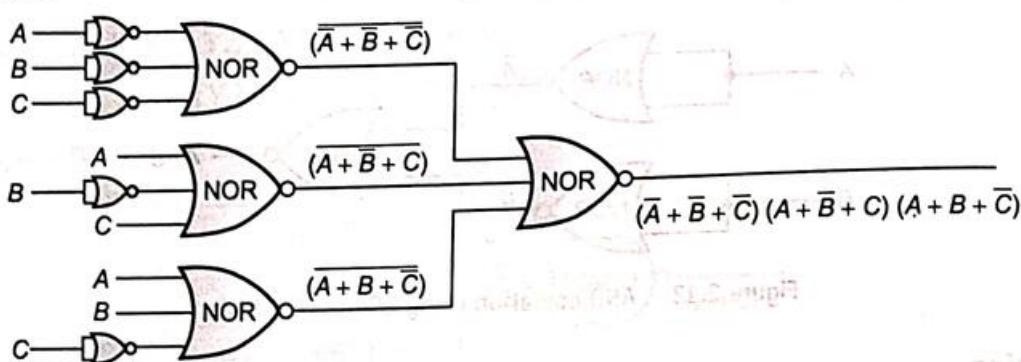
Example 2.18. Represent $(X + Y)(Y + Z)(Z + X)$ in NOR-to-NOR form.

Solution. $(X + Y)(Y + Z)(Z + X) = (X \text{ NOR } Y) \text{ NOR } (Y \text{ NOR } Z) \text{ NOR } (Z \text{ NOR } X)$



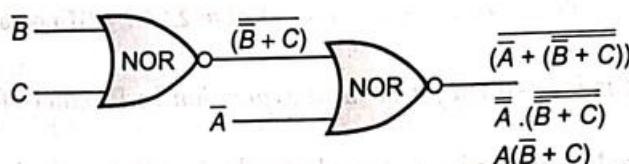
Example 2.19. Represent $(\bar{A} + \bar{B} + \bar{C})(A + \bar{B} + C)(A + B + \bar{C})$ in NOR-to-NOR logic network.

Solution.



Example 2.20. Show $A(\bar{B} + C)$ using NOR gates only.

Solution.



2.5 Applications of Logic Gates

The design and maintenance of digital computers are greatly facilitated by the use of boolean algebra and logic circuits. Logic networks are designed making use of logic gates.

Logic gates have several applications to the computers. These are used in following useful applications :

1. ADDERS
 - Half Adder
 - Full Adder
2. Encoders
3. Decoders

and many more. However, we are going to cover Adders only, as per the syllabus.

2.5.1 Adders

We are familiar with ALU (Arithmetic Logic Unit) which performs all arithmetic and logic operations. But ALUs do not process decimal numbers ; they process binary numbers. Thus adders in ALU also work on binary numbers.

Before we get into details, all you need to know is *rules for Binary addition, which are given below:*

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

$$1 + 1 + 1 = 11$$

See, the result is having two bits, right bit gives you the *Sum* and left bit gives you the *Carry*.

Half Adder

It is a logic circuit that adds two bits. It produces the outputs : SUM and CARRY. The boolean equations for SUM and CARRY are :

$$\text{SUM} = X \oplus Y = \bar{X}Y + X\bar{Y}$$

$$\text{CARRY} = X \cdot Y$$

SUM is X XOR Y ; and CARRY is X AND Y.

Therefore, SUM produces 1, when X and Y are different and CARRY is 1 when X and Y both are 1's. Truth Table for Half Adder is given in (Table 2.21).

Table 2.21 Truth Table for half adder

X	Y	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Logic circuit for Half adder has been given in Fig. 2.16.

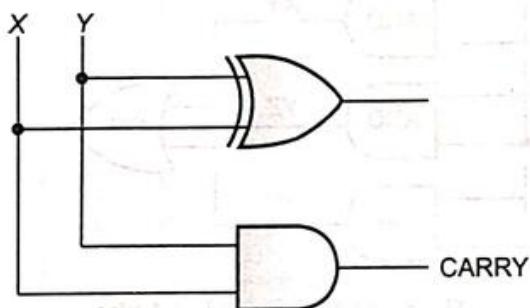


Figure 2.16 2-input Half adder.

Applications of Half adder are limited as only two bits can be added. Instead we need a circuit that can add three bits at a time.

Full Adder

It is a logic circuit that can add three bits. It produces two outputs : SUM and CARRY. The boolean equations for SUM and CARRY are : (Refer to Truth table 2.21)

$$\text{SUM} = X \oplus Y \oplus C = (\bar{X}\bar{Y}Z + \bar{X}YZ + XY\bar{Z} + XYZ)$$

$$\text{CARRY} = XY + YZ + ZX \quad (\text{Simplified from expression } \bar{X}YZ + X\bar{Y}Z + XY\bar{Z} + XYZ)$$

i.e., SUM equals X XOR Y XOR Z

CARRY equals XY OR YZ OR ZX

Therefore, SUM produces 1 when input is containing odd number of 1's and CARRY is 1 when there are two or more 1's in input.

Truth Table for Full Adder is given below (Table 2.22).

Table 2.22 Truth Table for full adder

X	Y	Z	Carry	Sum
0	0	0	0	0
0	0	1	0	1 $\bar{X}\bar{Y}Z$
0	1	0	0	1 $\bar{X}YZ$
0	1	1	1 $\bar{X}YZ$	0
1	0	0	0	1 $X\bar{Y}\bar{Z}$
1	0	1	1 $X\bar{Y}Z$	0
1	1	0	1 $XY\bar{Z}$	0
1	1	1	1 XYZ	1 XYZ

Logic circuit for Full Adder has been given in Fig. 2.17.

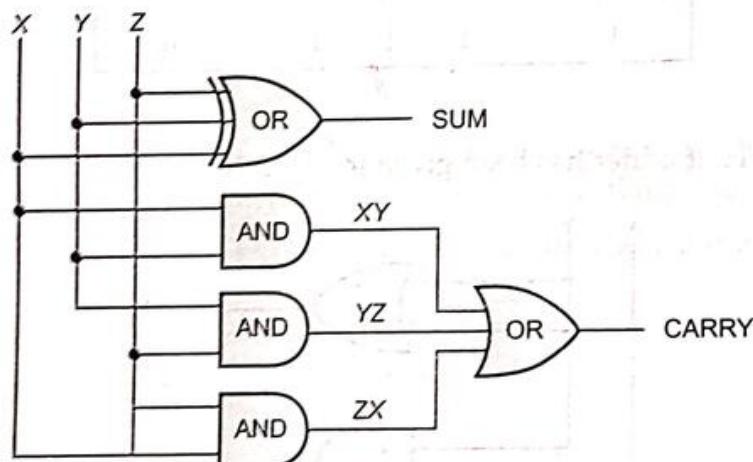


Figure 2.17 3-input Full Adder.

Let Us Revise

- ↗ The NOT gate or inverter is a gate with only one input signal and one output signal; the output state is always the opposite of the input state.
- ↗ The OR gate has two or more input signals but only one output signal. If any of the input signals is 1 (high), the output signal is 1 (high).
- ↗ The AND gate can have two or more input signals and produce an output signal. When all the inputs are 1 i.e., high then the output is 1 otherwise output is 0.
- ↗ The NOR gate has two or more input signals but only one output signal. If all the inputs are 0 (i.e., low), then the output is 1 (high).
- ↗ The NAND gate has two or more input signals but only one output signal. If all of the inputs are 1 (high), then the output produced is 0 (low).
- ↗ XOR gate has two or more inputs but one output. It produces output 1 for only those input combinations that have odd number of 1's.

- ❖ XNOR gate is Inverted XOR gate. It produces 1 (high) output when the Input combination has even number of 1's.
- ❖ Design rules for NAND-to-NAND logic Network : (I) derive simplified sum-of-products expression. (II) draw a circuit diagram using AND, OR gates. (III) Just replace AND and OR gates with NAND gates.
- ❖ Design rule for NOR-to-NOR logic Network (I) derive simplified P-O-S expression (II) draw a circuit using OR, AND gates. (III) replace OR and AND gates with NOR gates.
- ❖ Logic network have several applications, some of them are : Adders (half adder, full adder), Encoders, Decoders etc.
- ❖ Half adder is logic circuit that adds two bits.
- ❖ Full adder is logic circuit that adds three bits.

Solved Problems

1. Construct a truth table for the expression \bar{a} . What single term is the expression equivalent to ?

Solution.

a	\bar{a}	$\bar{\bar{a}}$
0	1	0
1	0	1

Comparing the columns a and $\bar{\bar{a}}$, we find that $\bar{\bar{a}} = a$. Hence the expression \bar{a} is equivalent to a .

2. Prove that $x + yz = (x + y)(x + z)$

Solution.

x	y	z	yz	$x + yz$	$(x + y)$	$(x + z)$	$(x + y)(x + z)$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Comparing the columns, we find that $x + yz = (x + y)(x + z)$. Hence proved.

3. Verify using truth table that $(x + y)' = x' \cdot y'$.

Solution. As it is a 2-variable expression, truth table will be as follows :

x	y	$x + y$	$(x + y)'$	x'	y'	$x' \cdot y'$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Comparing the columns $(x + y)'$ and $x' \cdot y'$, both of the columns are identical, hence verified.

4. From the premises $p \Rightarrow q$ and $q \Rightarrow p$, conclude $\bar{q} + pq$.

Solution. Given premises are :

$$P_1: p \Rightarrow q$$

$$P_2: q \Rightarrow p$$

Conclusion (C) to be drawn : $\bar{q} + pq$, we have to find out

$$(P_1, P_2) \Rightarrow C.$$

p	q	$p \Rightarrow q$	$q \Rightarrow p$	P_1, P_2		\bar{q}	pq	C		$(P_1, P_2) \Rightarrow C$
				$(p \Rightarrow q) \cdot (q \Rightarrow p)$	$\bar{q} + pq$					
0	0	1	1	1	1	1	0	1	1	1
0	1	1	0	0	0	0	0	0	1	1
1	0	0	1	0	1	1	0	1	1	1
1	1	1	1	1	1	0	1	1	1	1

Since $(P_1, P_2) \Rightarrow C$ results into tautology. Hence concluded.

5. From premises $p \Rightarrow q$ and $q \Rightarrow p$, conclude $\bar{q} + pq$ algebraically.

Solution. Given premises are :

$$P_1: p \Rightarrow q \text{ and } P_2: q \Rightarrow p$$

Conclusion (C) to be drawn : $\bar{q} + pq$.

We have to find out $(P_1, P_2) \Rightarrow C$

$$= [(p \Rightarrow q) \cdot (q \Rightarrow p)] \Rightarrow (\bar{q} + pq)$$

Carrying out conditional elimination, we get

$$= [(\bar{p} + q) \cdot (\bar{q} + p)] \Rightarrow (\bar{q} + pq)$$

Carrying out conditional elimination once again, we get

$$= [(\bar{p} + q) \cdot (\bar{q} + p)] + (\bar{q} + pq)$$

$$= (\bar{p} + q) + (\bar{q} + p) + \bar{q} + pq$$

[De Morgan's Law]

$$= (\bar{\bar{p}} \cdot \bar{q}) + (\bar{\bar{q}} \cdot \bar{p}) + \bar{q} + pq$$

[De Morgan's Law]

$$= p\bar{q} + q\bar{p} + \bar{q} + pq$$

$$= p\bar{q} + \bar{q} + q\bar{p} + pq$$

$$= \bar{q}(p+1) + q(\bar{p}+p)$$

$$= \bar{q} \cdot 1 + q \cdot 1$$

$[\because p+1=1; \bar{p}+p=1]$

$$= \bar{q} + q$$

$$= 1$$

$[\because \bar{q} + q = 1]$

Hence concluded.

6. Find out the equivalent expression for $(p \Leftrightarrow q) + (p \Rightarrow q)$, without having any conditional or bi-conditional.

Solution. Carrying out bi-conditional elimination, we get

$$p \Leftrightarrow q = (p \cdot q) + (\bar{p} \cdot \bar{q})$$

Carrying conditional elimination, we get

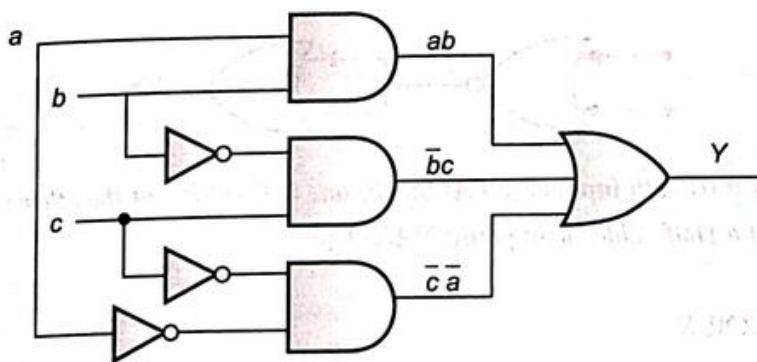
$$p \Rightarrow q = \bar{p} + q.$$

Thus, the equivalent expression is :

$$pq + \bar{p} \cdot \bar{q} + \bar{p} + q.$$

7. Draw logic circuit diagram for the following expression : $Y = ab + \bar{b}c + \bar{c}\bar{a}$

Solution.



8. Draw the simplified logic diagram using only NAND gates to implement :

$$\bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{x}yz + x\bar{y}z$$

Solution.

$$\bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{x}yz + x\bar{y}z = \bar{x}\bar{y}(\bar{z} + z) + \bar{x}yz + \bar{y}z$$

$$= \bar{x}\bar{y} \cdot 1 + \bar{x}yz + x\bar{y}z \quad (\bar{z} + z = 1)$$

$$= \bar{x}\bar{y} + \bar{x}yz + x\bar{y}z$$

$$= \bar{x}(\bar{y} + yz) + x\bar{y}z$$

$$= \bar{x}(\bar{y} + z) + x\bar{y}z$$

$$= \bar{x}\bar{y} + \bar{x}z + x\bar{y}z$$

$$= \bar{x}\bar{y} + z(\bar{x} + x\bar{y})$$

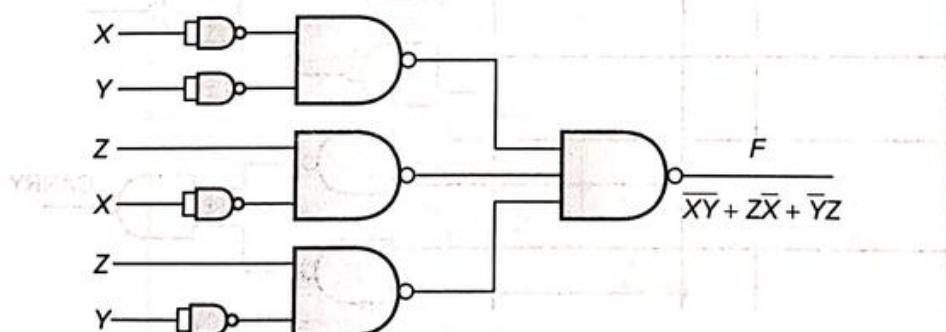
$$= \bar{x}\bar{y} + z(\bar{x} + \bar{y})$$

$$= \bar{x}\bar{y} + z\bar{x} + \bar{y}z$$

$$(\bar{y} + yz = \bar{y} + z \text{ Table 2.6})$$

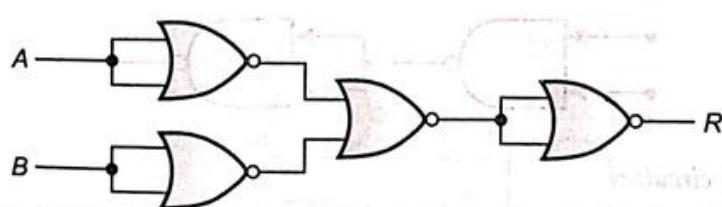
$$(\bar{x} + \bar{y} = \bar{x} + \bar{y} \text{ Table 2.6})$$

NAND-to-NAND circuit is



9. Draw the logic of NAND gate using NOR gates only.

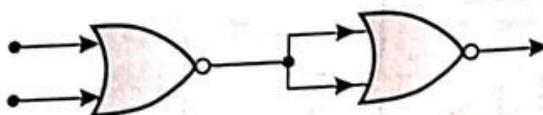
Solution.



R will give A NOR B

10. (a) Represent NOT using only NOR gate(s).

(b) Given the following circuit :

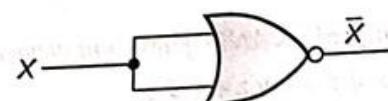


What is the output if (i) both inputs are FALSE (ii) one is FALSE and the other is TRUE ?

- (c) Draw the circuit of a Half Adder using only NAND gates.

Solution.

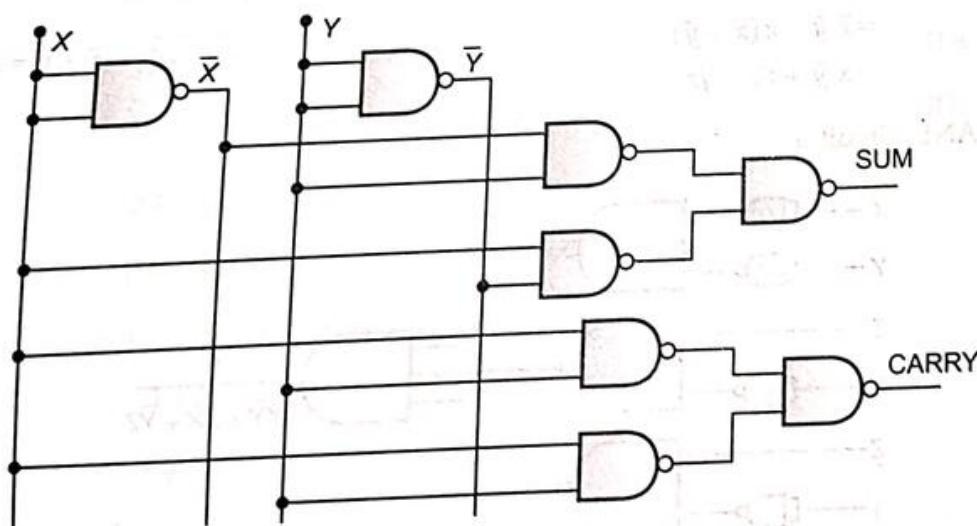
(a) $\bar{X} = \text{NOT } X = X \text{ NOR } X$



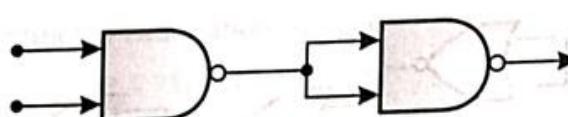
- (b) (i) False (ii) True

(c) In half adder Sum = $\bar{x}y + x\bar{y}$, Carry = $X \cdot Y$

x	y	Carry	Sum
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0



11. (a) Seven inverters are cascaded one after another. What is the output if the input is 1 ?



(b) Given the following circuit :

What is the output if (i) both inputs are FALSE (ii) one is FALSE and the other is TRUE ?

- (c) Derive the expression for a Full Adder.

Solution.

(a) 0

(b) (i) False (ii) False

(c) Full Adder It is a logic circuit that can add three bits. It produces two outputs : SUM and CARRY.
The boolean equations for SUM and CARRY are

$$\text{SUM} = X \oplus Y \oplus C, \\ \text{i.e., } \text{SUM equals } X \text{ XOR } Y \text{ XOR } Z$$

$$\text{CARRY} = XY + YZ + ZX \\ \text{CARRY equals } XY \text{ OR } YZ \text{ OR } ZX$$

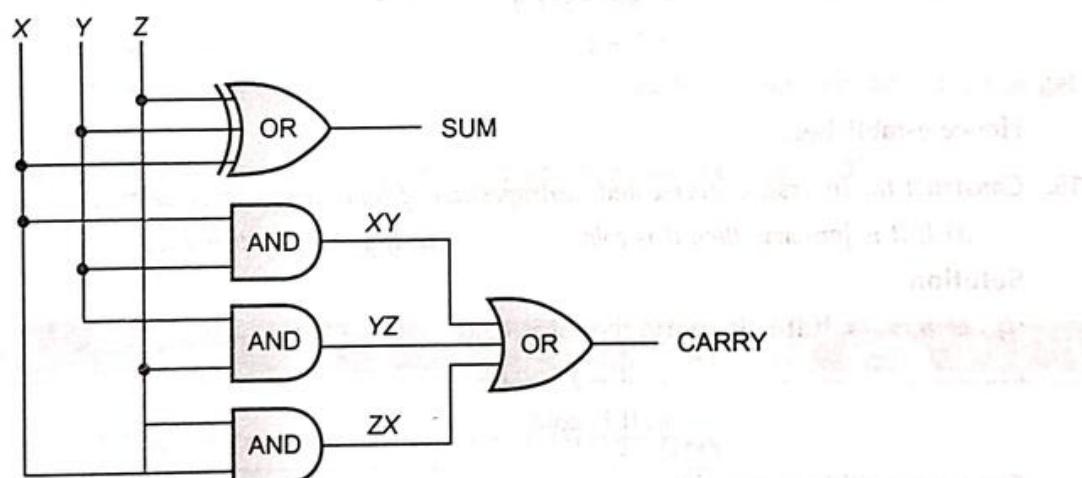
Therefore, SUM produces 1 when input is containing odd number of 1's and CARRY is 1 when there are two or more 1's in input.

Truth Table for Full Adder is given below :

Truth Table for full adder

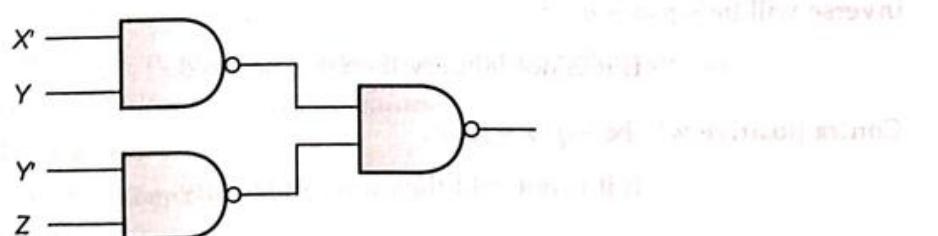
X	Y	Z	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Logic circuit for Full Adder has been given in following figure :



12. Represent the Boolean expression $X'Y + Y'Z$ with the help of NAND gates only.

Solution.



13. Given wffs are : $p \Rightarrow q$ and $\sim q$, show that $\sim p$ is a logical consequence of two preceding wffs.

Solution. We have to determine $[(p \Rightarrow q), \sim q] \Rightarrow \sim p$

p	q	$p \Rightarrow q$	$\sim q$	$(p \Rightarrow q) \cdot \sim q$	$\sim p$	$[(p \Rightarrow q) \cdot \sim q] \Rightarrow \sim p$
0	0	1	1	1	1	1
0	1	1	0	0	1	1
1	0	0	1	0	0	1
1	1	1	0	0	0	1

$[(p \Rightarrow q) \cdot \sim q] \Rightarrow \sim p$ is a tautology. Hence,

$$p \Rightarrow q$$

$$\frac{\sim q}{\sim p}$$

14. Solve the previous question algebraically.

Solution. We have to determine $[(p \Rightarrow q), \sim q] \Rightarrow \sim p$.

Carrying out conditional elimination, we get

$$\begin{aligned} &= [(\sim p + q) \cdot \sim q] \Rightarrow \sim p \\ &= (\sim p \cdot \sim q + q \cdot \sim q) \Rightarrow \sim p \\ &= (\sim p \cdot \sim q + 0) \Rightarrow \sim p \end{aligned} \quad [\because q \cdot \sim q = 0]$$

Carrying out conditional elimination once again, we get

$$\begin{aligned} &\sim(\sim p \cdot \sim q) + \sim p \\ &= \sim \sim p + \sim \sim q + \sim p \quad [\because \sim(\sim p \cdot \sim q) = \sim \sim p + \sim \sim q, \text{ De Morgan's Law}] \\ &= p + q + \sim p \\ &= p + \sim p + q \\ &= 1 + q \quad [\because p + \sim p = 1] \\ &= 1 \quad [\because 1 + q = 1] \end{aligned}$$

Hence established.

15. Construct the inverse, converse and contrapositive of conditionals given below :

(i) If it is January, then it is cold. (ii) If $y + 5 \neq 7$, then $y < 0$

Solution.

(i) Let $p \Rightarrow q$ = If it is January, then it is cold.

i.e.,

p : It is January

q : It is cold.

Converse will be $q \Rightarrow p$, i.e.,

If it is cold then it is January.

Inverse will be $\sim p \Rightarrow \sim q$, i.e.,

If it is not January then it is not cold.

Contra-positive will be $\sim q \Rightarrow \sim p$, i.e.,

If it is not cold then it is not January.

(ii) Let $a \Rightarrow b = \text{If } y + 5 \neq 7, \text{ then } y < 0, \text{ i.e.,}$

$$a: y + 5 \neq 7$$

$$b: y < 0$$

Converse will be $b \Rightarrow a, \text{ i.e.,}$

If $y < 0$ then $y + 5 \neq 7$

Inverse will be $\sim a \Rightarrow \sim b, \text{ i.e.,}$

If $y + 5 = 7$ then $y \not< 0$

Contra-positive will be $\sim b \Rightarrow \sim a, \text{ i.e.,}$

If $y \not< 0$ then $y + 5 = 7$.



Glossary

AND gate A logic circuit whose output is 1 (*high*) only when all inputs are 1 (*high*).

Connective Operator joining simple statements into compounds.

Gate A logic circuit with one or more input signals but only one output signal.

NAND gate Logically means an AND gate followed by an inverter. All inputs must be 1 (*high*) to get a 0 (*low*) output.

NOR gate Logically means an OR gate followed by an inverter. All inputs must be 0 (*low*) to get a 1 (*high*) output.

NOT gate A gate with one input and one output signal, output being complement of the input.

OR gate A logic circuit whose output is 1 (*high*) only when one or more inputs are 1 (*high*).

Proposition Elementary atomic sentence that may either be true or false.

Syllogism Logical process of drawing conclusion from given premises.

Truth table A table of combinations showing all input and output possibilities for a boolean expression.

Truth value Truth or falsity of a proposition.

XNOR gate Logically means an XOR gate followed by an inverter. Inputs must have even number of 1's to get a 1 (*high*) output.

XOR gate A gate which produces 1 (*high*) output only when inputs have even number of 1's.



Assignments

TYPE A : VERY SHORT ANSWER QUESTIONS

- What is proposition ?
- What do you mean by contingency, tautology and contradiction ?
- What is a logic gate ? Name the three basic logic gates.
- Which gates implement logical addition, logical multiplication and complementation ?
- Write the precedence order of evaluation of logical operators.
- What is the other name of NOT gate ?
- What is a truth table ? What is the other name of truth table ?
 - $A + 0 = ?$
 - $A + 1 = ?$
 - $A \cdot 0 = ?$
 - $A \cdot 1 = ?$

8. According to which law following expressions are true :

$$X + (Y + Z) = (X + Y) + Z \text{ and } X(YZ) = (XY)Z$$

9. Which law states that $X(Y + Z) = XY + XZ$?

10. What is the following property of called ? $X \cdot (Y + Z) = X \cdot Y + X \cdot Z$

11. What is the following property of called ? $X + XY = X$

12. Rule $\overline{X + Y} = \overline{X} \cdot \overline{Y}$ and Rule $\overline{X \cdot Y} = \overline{X} + \overline{Y}$ are known as ?

13. Which of the following is/are incorrect? Write the correct forms of the incorrect ones :

$$(a) A + A' = 1 \quad (b) A + 0 = A \quad (c) A \cdot 1 = A \quad (d) AA' = 1$$

$$(e) A + AB = A \quad (f) A(A + B)' = A \quad (g) (A + B)' = A' + B \quad (h) (AB)' = A' B'$$

$$(i) A + 1 = 1 \quad (j) A + A = A \quad (k) A + A' B = A + B \quad (l) X + YZ = (X + Y)(X + A)$$

14. What is a connective ?

15. Name the process for the compounds given below :

$$(i) p + q \quad (ii) p \cdot q \quad (iii) p \Rightarrow q \quad (iv) \sim p \quad (v) p \Leftrightarrow q$$

16. What are contingencies ?

17. Form the converse, inverse and contrapositive of condition $p \Rightarrow q$.

18. What is syllogism ?

19. Why are NAND and NOR gates called Universal gates ?

20. Which gates are called Universal gates and why ?

21. Draw a logic circuit diagram using NAND or NOR gates only to implement the Boolean function $F(a, b) = a'b' + ab$

22. How does half adder differ from full adder ?

23. What is inverted AND gate called ? What is inverted OR gate called ?

24. When does an XOR gate produce a high output ? When does an XNOR gate produce a high output ?

25. Write some applications of logic networks.

TYPE B : SHORT ANSWER QUESTIONS

1. Given the following simple propositions :

p = It is raining

q = It is not a sunny day.

Construct the compound sentences for the following expressions :

$$(i) q' \quad (ii) p \cdot q \quad (iii) p + q \quad (iv) p \Rightarrow q \quad (v) p \Leftrightarrow q \\ (vi) p' + q' \quad (vii) p' \cdot q' \quad (viii) \sim p \Rightarrow q \quad (ix) \sim p \Leftrightarrow \sim q \quad (x) (p + q) \Rightarrow (p \cdot q)$$

2. (a) If x represents "I like coffee" and y represents "I like tea" then write in symbolic form :

(i) I like coffee and tea ; (ii) I like coffee but not tea ;

(iii) It is false that I don't like coffee or tea ; (iv) Neither I like coffee nor tea ;

(v) Either I like coffee or I do not like coffee but like tea.

(b) For what statement does $\sim x \wedge y$ stand for ?

3. If s stands for the statement "I will not go to school", and t for the statement, "I will watch a movie", then what does $\sim s + t$ stand for ?

4. Establish the following using truth tables :

$$(i) \sim(a \Rightarrow b) = p \cdot \sim q \quad (ii) a \Rightarrow b = \sim a \Rightarrow \sim b \quad (iii) (a \Rightarrow b) \cdot (b \Rightarrow a) = a \Leftrightarrow b$$

$$(iv) (\sim a + b) \wedge (\sim b + a) = a \Leftrightarrow b \quad (v) (p \Rightarrow r) \cdot (q \Rightarrow r) = (p + q) \Rightarrow r$$

5. What will be the result of following compounds if given inputs are
 (i) $x = 0, y = 1$; (ii) $x = 1, y = 1$; (iii) $x = 1, y = 0$
 (a) $y \wedge x$ (b) $y \vee x$ (c) $\sim x \vee y$ (d) $x \wedge \sim y$ (e) $(\sim x \wedge \sim y) \vee \sim y$
6. Determine the converse, inverse and contrapositive for conditional (i) $x \Rightarrow y$, (ii) $p \Rightarrow q$ where
 x : It is raining, y : I am enjoying it.
 $p: 2 + 3 \neq 6$, $q: a > 0$

[Hint. Refer to solved problem 14]

7. Find out which of the following are tautologies and which of them are contradictions ?
 (a) $p \cdot q (p + q)$ (b) $(p \Rightarrow q) \cdot p$
 (c) $(p \cdot q) \Rightarrow (\sim p \cdot \sim q)$ (d) $[(p \Rightarrow q) \Leftrightarrow \sim q \Rightarrow (\sim p \wedge \sim q)]$
 (e) $(p + q) \cdot (p' \cdot q')$ (f) $[(p \Rightarrow q) \wedge (q \Rightarrow r)]$
8. Given that : $p: 2 + 3 = 5$
 $q: 2 \times 3 = 6$

Now construct the truth table for following compounds :

- (i) if $2 + 3 = 5$, then $2 \times 3 = 6$ (ii) if $2 + 3 \neq 5$, then $2 \times 3 = 6$
 (iii) if $2 + 3 = 5$, then $2 \times 3 \neq 6$ (iv) if $2 + 3 \neq 5$, then $2 \times 3 \neq 6$

9. Establish the validity for the following :

$$\begin{array}{c} p \Rightarrow q \\ p \Rightarrow r \\ \hline \sim r \Rightarrow q \end{array}$$

10. Establish that

$$\begin{array}{c} a + b \\ \hline \sim a \Rightarrow b \\ \hline \sim b \end{array}$$
11. Establish that

$$\begin{array}{c} a \Rightarrow b \\ b \Rightarrow c \\ \hline a \\ \hline c \end{array}$$
12. Establish that

$$\begin{array}{c} x \Leftrightarrow \sim y \\ y \Rightarrow z \\ \hline z \end{array}$$

13. Draw the conclusions from the following premises :

P1 : John is a father.

P2 : If John is a father then John has a child.

[Hint. Modus ponens]

14. Determine whether each of the following sentences is
 (a) satisfiable (b) contradictory (c) valid
- | | |
|--|--|
| $S_1 : (p \& q) \vee \sim (p \& q)$ | $S_2 : (p \vee q) \rightarrow (p \& q)$ |
| $S_3 : (p \& q) \rightarrow r \vee \sim q$ | $S_4 : (p \vee q) \& (p \vee \sim q) \vee p$ |
| $S_5 : p \rightarrow q \rightarrow \sim p$ | $S_6 : p \vee q \& \sim p \vee \sim q \& p$ |

15. Find the meaning of the statement :
 $(\sim p \vee q) \& r \rightarrow s \vee (\sim r \& q)$

for each of the interpretations given below :

- (a) I_1 : p is true, q is true, r is false, s is true.
 (b) I_2 : p is true, q is false, r is true, s is true.

16. What do you understand by 'truth value' and 'proposition'? How are these related?
17. What do you understand by 'logical function'? What is its alternative name? Give examples for logical functions.
18. What is meant by tautology and contradiction? Prove that $1 + Y$ is a tautology and $0 \cdot Y$ is a contradiction.

19. What is a truth table ? What is its significance ?
20. Verify using truth table that $X + XY = X$ for each X, Y in {0, 1}.
21. Verify using truth table that $(X + Y)' = X'Y'$ for each X, Y in {0, 1}.
22. Give truth table for the Boolean Expression $(X + Y')$.
23. Draw the truth table for the following equations : (a) $M = N(P + R)$ (b) $M = N + P + N\bar{P}$
24. Using truth table, prove that $AB + BC + C\bar{A} = AB + C\bar{A}$
25. State the distributive laws. How do they differ from the distributive laws of ordinary algebra ?
26. Prove the idempotence law with the help of a truth table.
27. Prove the complementarity law with the help of a truth table.
28. Give the truth table proof for distributive law.
29. Prove the following :
- (i) $A(B + C + D) = A$ (ii) $\bar{A}\bar{A}B = A + \bar{B}$
 - (iii) $(x + y + z)(\bar{x} + y + z) = y + z$ (iv) $AB + BC + BCD = A(\bar{A}D + C)$
30. Draw logic circuit diagrams for the following :
- (i) $xy + x\bar{y} + \bar{x}z$ (ii) $(A + B)(B + C)(\bar{C} + \bar{A})$ (iii) $\bar{A}B + BC$ (iv) $xyz + \bar{x}yz$
31. Design a circuit (3 input) which gives a high input, when there is even number of low inputs.
32. Design a circuit (3 input) which given a high input only when there is even number of low or high inputs.
33. Design a logic circuit to realize the Boolean function $f(x, y) = x \cdot y + x' \cdot y'$
34. Draw the logic circuit for this boolean equation :
- $$y = \bar{A}\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + A\bar{B}C\bar{D} + ABC\bar{D}$$
35. Draw the AND-OR circuit for $y = A\bar{B}\bar{C}\bar{D} + A\bar{B}\bar{C}D + ABCD$
36. Convert the above circuit into NAND-to-NAND logic circuit.
37. Why are NAND and NOR gates more popular ?
38. Draw the logical circuits for the following using NAND gates only :
- (i) $xy + x\bar{y} + xyz$ (ii) $ABC + A\bar{B}\bar{C} + \bar{A}BC$
39. Draw the logical circuits for the following using NOR gates only :
- (i) $(X + Y)(\bar{X} + Y)(\bar{X} + \bar{Y})$ (ii) $(X + Y + Z)(X + \bar{Y} + \bar{Z})$
40. (a) State De Morgan's Laws. Verify them using truth tables.
- (b) Prove $(A + B)(A' + C) = (A + B + C)(A + B + C')$.
- (c) Give the truth table for a Full-adder.
- (d) Draw the circuit diagram for the Boolean function $F(X, Y, Z) = (X' + Y)(Y' + Z)$ using NOR gates only.
41. (a) State the distributive law. Verify the law using truth table.
- (b) Prove $x + x'y = x + y$.
- (c) Draw the logic circuit for a half-adder.
- (d) Represent the Boolean expression $(x + y)(y + z)(z + x)$ with the help of NOR gates only.
42. (a) State the Idempotence law. Verify the law using truth table.
- (b) Prove $(x + y)(x + z) = x + xz$ algebraically.
- (c) Draw the logic circuit for a Full Adder.
- (d) Represent the Boolean expression $yz + xz$ with the help of NAND gates only.
43. (a) State Absorption Laws. Verify one of the Absorption laws using a truth table.
- (b) Represent the Boolean expression $X + Y \cdot Z'$ with the help of NAND gates only.

General OOP Concepts

In This Chapter

- 3.1 Introduction
- 3.2 Evolution of Software
- 3.3 Basic Concepts of OOP

3.1 Introduction

With the rapidly changing world and the highly competitive and versatile nature of industry, the operations are becoming more and more complex. In view of the increasing complexity of software systems, the software industry and software engineer continuously look for the new approaches to software design and development. The increased complexity had become the chief problem with computer programs in traditional languages. Large programs, because of this complexity, are more prone to errors, and software errors can be expensive and even life-threatening. The most adopted and popular programming approach, structured programming approach, failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs. The programming approach, Object-Oriented Programming (OOP), offers a new and powerful way to cope with this complexity. Its goal is clearer, more reliable, more easily maintained programs. This chapter introduces general OOP concepts that the traditional languages like C, Pascal, COBOL and BASIC lack in and the new generation Object-Oriented Languages support.

3.2 Evolution of Software

A program serves the purpose of commanding the computer. The efficiency and usefulness of a program depends not only on proper use of commands but also on the programming language it is written in. The two major types of programming languages : *Low Level languages* and *High Level languages* offer different features of programming.

Low Level Languages

Low Level Languages (*i.e., machine language and assembly language*) are machine-oriented and require extensive knowledge of computer circuitry. *Machine language*, in which instructions are written in binary code (using 1 and 0), is the only language the computer can execute directly. *Assembly language*, in which instructions are written using symbolic names for machine operations (*e.g.*, READ, ADD, STORE etc.) and operands, makes programming less tedious than machine language programming. However, assembly program is then converted into machine language using *assembler* software.

High Level Languages

High Level Languages, (*HLLs*), on the other hand, offer English like keywords, constructs for sequence, selection (decision) and iteration (looping) and use of variables and constants. Thus it is very easy to program with such languages compared to low level languages. The programs written in HLLs are converted into machine language using compiler or interpreter as a computer can work with machine language only.

A programming language should serve *two* related purposes :

- (i) it should provide a vehicle for the programmer to specify actions to be executed and
- (ii) it should provide a set of concepts for the programmer to use when thinking about what can be done.

The first aspect ideally requires a language that is "*close to the machine*", so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The second aspect ideally requires a language that is "*close to the problem to be solved*" so that the concepts of a solution can be expressed directly and concisely.

The *low level* languages serve only the first aspect *i.e.*, they are close to the machine and the *high level* languages serve only the second aspect *i.e.*, they are close to the programmer. However, the languages 'C' and 'C+' serve both the aspects, hence can be called as '*middle level languages*'.

3.2.1 Programming Paradigms

By paradigm, one means a way of thinking or doing things.

Since the invention of the computer, many programming approaches have been tried such as procedural programming, modular programming, structural programming etc. The primary motivation in each case has been the concern to handle the increasing complexity of programs that are reliable and maintainable.

Let us discuss these different programming paradigms and key language mechanisms necessary for supporting them.

Paradigm means organizing principle of a program. It is an approach to programming.

Procedural Programming

A program in a procedural language is a list of instructions where each statement tells the computer to do something. The focus is on the processing, the algorithm needed to perform the desired computation.

This paradigm is :

Decide which procedures you want ; use the best algorithms you can find.

Languages support this paradigm by providing facilities for *passing arguments to functions* (subprograms) and *returning values from functions* (subprograms).

In procedural paradigm, the emphasis is on doing things. What happens to the data ? Data is, after all, the reason for a program's existence. The important part of an inventory program isn't a function that displays or checks data ; it is the inventory data itself. Yet data is given second-class status while programming.

Modular Programming

With the increase in program size, a single list of instructions becomes unwieldy. Thus a large program is broken down into smaller units *i.e.*, functions (sub-programs). The idea of breaking a program into functions can further be extended by grouping a number of functions together into a larger entity called a module, but the principle is similar : grouping of components that carry out specific tasks.

A set of related procedures with the data they manipulate is called a **module**.

This programming paradigm, also known as 'data-hiding principle' states :

Decide which modules you want ; partition the program so that data is hidden in **modules**.

Where there is no grouping of procedures with related data, the procedural programming style suffices. The techniques for designing 'good procedures' still apply on each member procedure of a module.

In modular programming, since many modules (or functions) access the same data, the way the data is stored becomes critical. The arrangement of the data can't be changed without modifying all the functions that access it.

Another problem associated with procedural and modular programming is that their chief components – functions etc. do not model the real world very well. For instance, a procedural program for library maintenance aims at the operations *Issue*, *Return* etc. whereas the real world entities are Books. But 'Books' are given second-class status in the program. We will understand this problem more clearly in the next paradigm – the object oriented paradigm.

The Object Oriented Programming

To understand this most recent concept among programming paradigms, let us take an example. We have to prepare lot of dishes that involve baking ; for instance, cake, biscuits, pie, pastries, buns etc. We have to write programs for it. Using procedural programming paradigm, we'll have to write separate programs for every recipe and each program will be containing instructions for operating oven. Ovens are used to make a lot of different dishes. We don't want to create a new oven every time we encounter a new recipe. Having solved a

problem once, it would be nice to be able to *reuse* the solution in future programs. But the procedural programming paradigm suffers from this drawback. Though functional software reuse is theoretically possible, but it hasn't worked very well in practice. The large amount of interaction between conventional functions and its surroundings makes reusing them difficult.

However, if the same problem is solved using *Object-Oriented* approach, it'll not only be less complex but also make software reuse feasible and possible. The *object-oriented* approach views a problem in terms of *objects* involved rather than *procedure* for doing it.

Object is an identifiable entity with some characteristics and behaviour.

For instance, we can say '*Orange*' is an *object*. Its characteristics are : it is *spherical shaped*, its *colour is orange* etc. Its behaviour is : it is *juicy citrus* and it tastes *sweet-sour*. While programming using OOP approach, the characteristics of an object are represented by its data and its behaviour is represented by its functions associated. Therefore, *in OOP programming object represents an entity that can store data and has its interface through functions*.

The above mentioned problem of baking dishes, in OOP approach, will be viewed in terms of objects involved (*i.e.*, OVEN) and its interface (*i.e.*, the functions representing its working). This not only results in simpler program design but also reduces software cost.

To understand this concept more clearly, let us consider another example. Let us simulate traffic flow at a red light crossing.

As you know, procedural programming paradigm focuses on the procedures or the working action.

Using procedural programming paradigm, the above said problem will be viewed in terms of working happening in the traffic-flow *i.e.*, moving, halting, turning etc. The OOP paradigm, however, aims at the objects and their interface. Thus in OOP approach, the traffic-flow problem will be viewed in terms of the objects involved. The objects involved are : cars, trucks, buses, scooters, auto-rickshaws, taxis etc.

With this elaboration, the concept must be clear enough to proceed for the OOP paradigm definition. But before that let us understand another very important term *class*.

A **class** is a group of objects that share common properties and relationships.

In the above example, cars have been identified as objects. They have characteristics like : steering wheel, seats, a motor, brakes etc. and their behaviour is their mobility. Car, however, is not an object, it is a class (compare with the definition). *Opel Astra* having reg. no. 5523 is an object that belongs to the class '*car*'. '*Car*' is a subclass of another class '*automobiles*' which again is a subclass of '*vehicles*'. '*Object*' is an instance of '*class*'. For example, we can say '*bird*' is a class but '*parrot named Mithu*' is an object.

Now, after understanding the basic terminology, we can define object-oriented programming (OOP) paradigm as :

Decide which classes and objects are needed ; provide a full set of operations for each class.

This entire chapter focuses on OOP paradigm and in the coming chapters, you'll be implementing some¹ of these concepts using programming language Java.

1. Some concepts such as Inheritance shall be covered in class XII.

3.3 Basic Concepts of OOP

The object-oriented programming has been developed with a view to overcome the draw-backs of conventional programming approaches. The OOP approach is based on certain concepts that help it attain its goal of overcoming the drawbacks or shortcomings of conventional programming approaches.

These general concepts of OOP are given below :

- ◆ Data Abstraction
- ◆ Data Encapsulation
- ◆ Modularity
- ◆ Inheritance
- ◆ Polymorphism

Now we'll discuss these concepts in little details to understand their meaning.

3.3.1 Data Abstraction

Abstraction is the concept of simplifying a real world concept into its essential elements.

To understand abstraction, let us take an example. You are driving a car. You only know the essential features to drive a car e.g., gear handling, steering handling, use of clutch, accelerator, brakes etc. etc. But while driving do you get into internal details of car like wiring, motor working etc.? You just change the gears or apply the brakes etc. What is happening inside is *hidden* from you. This is *abstraction* where you only know the essential things to drive a car without including the background details or explanations. Take another example of 'switch board'. You only press certain switches according to your requirement. What is happening inside, how it is happening etc. you needn't know. Again this is abstraction, you know only the essential things to operate on switch board without knowing the background details of switchboard.

Abstraction refers to the act of representing essential features without including the background details or explanations.

3.3.2 Encapsulation

Encapsulation is the most fundamental concept of OOP. It is the way of combining both *data* and *the functions that operate on that data* under a single unit.

The wrapping up of data and functions (that operate on the data) into a single unit (called class) is known as **encapsulation**.

The only way to access the data is provided by the functions (that are combined along with the data). These functions are called *member functions* or methods in Java. The data cannot be accessed directly. If you want to read a data item in an object (an instance of the class), you call a member function in the object. It will read the item and return the value to you. You can't access the data directly. The data is hidden, so it is safe from accidental alteration. Data and its functions are said to be encapsulated into a single entity.

Let us now consider an analogy to encapsulation. In a big company, there are so many departments, *Sales, Accounts, Payroll, Purchase, Production* etc. Each department has its own personnel that maintain its data. Suppose an employee in the production dept. wants to know how much raw material has been purchased for the next month. The production dept employee would not be allowed to himself go through the purchase dept. data files. Rather

he'll have to issue a memo to the 'purchase' requesting for the required information. Then some employee of the 'purchase' dept. will go through the purchase data files and send the reply with the asked information. This practice ensures that the data is accessed accurately and that it is not corrupted by inept outsiders. Therefore, we can say here '*Department data and department employees are encapsulated into a single entity, the department*'. In the same way, objects provide an approach to program organization while helping to maintain the integrity of the program data (see Fig. 3.1). Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT). 'Data types' because these can be used to create objects of its own type.

Abstraction and encapsulation are complementary concepts : *abstraction* focuses upon the observable behaviour of an object, whereas *encapsulation* focuses upon the implementation

that gives rise to this behaviour. Encapsulation is most often achieved through *information hiding*, which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics, the structure of an object is hidden, as well as the implementation of its methods. Only the essential characteristics of object are visible. In the above example of departments, only the essential information of departments are visible viz. *dept-name, dept-head, no-of-employees* etc. The secret information which is not essential is hidden viz. *dept-profit/loss, stock* etc. This information is made available only if the request is made through proper channel e.g., issuing a memo etc.

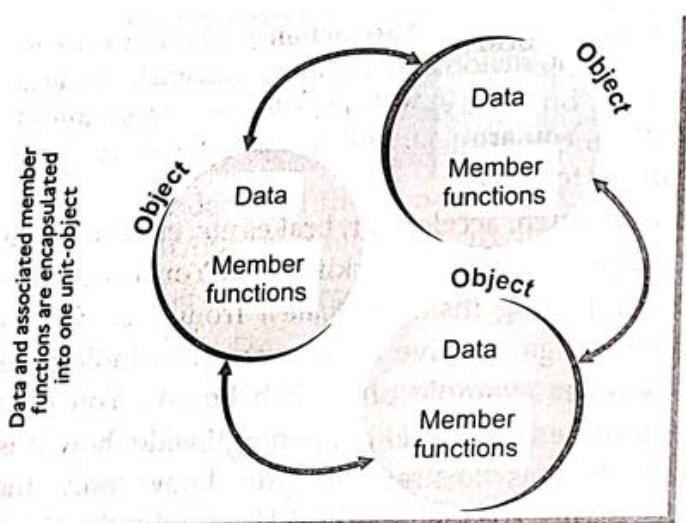


Figure 3.1 The OOP approach
(Data Abstraction and Encapsulation.)

3.3.3 Modularity

The act of partitioning a program into individual components is called modularity. The justification for partitioning a program is that

- (i) it reduces its complexity to some degree and
- (ii) it creates a number of well-defined, documented boundaries within the program.

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

A module is a separate unit in itself. It can be compiled separately though it has connections with other modules. Modules work hand in hand in order to achieve the program's goal.

For example, you must have seen a complete *music system*. Let us assume that our program represents the music system. The music system comprises of *speakers, cassette-player, record-player, cd-player, tuner* etc. Similarly our program can be divided into various modules each representing *speakers, cassette-player, etc.* See each module is a complete unit in itself yet it works in accordance with other modules in order to achieve one single goal i.e., music.

In object-oriented languages, classes and objects form the logical structure of a system. We place them in modules to produce the system's physical architecture. Especially for larger applications, in which we may have many hundreds of classes, the use of modules is essential to help manage complexity. Various OOP languages support modularity in diverse ways. For example, modularity in Java is implemented through packages. The traditional practice in the Java community is to place module interface in packages. The program files that depend upon package-contents can use them through import command. (It will become more clear to you when you start writing Java programs especially packages).

3.3.4 Inheritance

Understanding inheritance is critical to understand the whole point behind object-oriented programming.

For instance, we are humans. We inherit from the class '*Human*' certain properties, such as ability to speak, breathe, eat, drink etc. etc. But these properties are not unique to humans. The class '*Human*' inherits these properties from the class '*Mammal*' which again inherits some of its properties from another class '*Animal*'.

Inheritance is the capability of one class of things to inherit capabilities or properties from another class.

We take our same old example of cars. The class '*Car*' inherits some of its properties from the class '*Automobiles*' which inherits some of its properties from another class '*Vehicles*'. The capability to pass down properties is a powerful one. It allows us to describe things in an economical way. The object-oriented languages express this inheritance relationship by allowing one class to inherit from another. Thus a model developed by languages is much closer to the real world. The principle behind this sort of division is that each subclass shares common characteristics with the class from which it is derived (Fig. 3.2).

- ❖ '*Automobiles*' and '*Pulled Vehicles*' are *subclasses* of '*Vehicles*'. '*Vehicles*' is *base class* or *super class* of '*Automobiles*' and '*Pulled Vehicles*'.
- ❖ '*Car*' and '*Bus*' are *subclasses* or *derived classes* of '*Automobiles*'.
- ❖ '*Automobiles*' is the *base-class* or *super class* of '*Car*' and '*Bus*'.

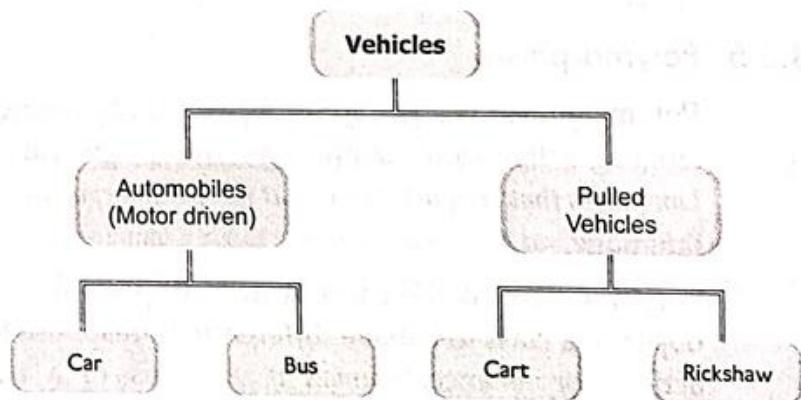


Figure 3.2 Property Inheritance.

Why Inheritance ?

There are several reasons why inheritance was introduced into OO languages. We are discussing here some major reasons behind introduction of inheritance.

1. One major reason behind this is the capability to express the inheritance relationship which makes it ensure the closeness with the real-world models (remember the real world stuff mentioned above).

2. Another reason is the idea of reusability. Inheritance allows the addition of additional features to an existing class without modifying it. One can derive a new class (sub-class) from an existing one and add new features to it. Suppose we have a class 'student', and we need to add a new class called 'GraduateStudent'. We derive the new class 'GraduateStudent' from the existing class 'Student' and then all we really need to add the extra features to 'GraduateStudent' that describe the differences between students and graduate student. Notice the reduction in amount of typing and efforts.

3. One reason is the transitive nature. For example, if a new class 'GraduateStudent' has been declared as a sub-class of 'Student' which itself is a sub-class of 'Person' then 'GraduateStudent' must also be a 'Person' i.e., inheritance is transitive. If a class A inherits properties of another class B, then all subclasses of A will automatically inherit the properties of B.

This property is called *transitive* nature of inheritance.

This carries a benefit with it. Suppose we inherit class B from existing class A. The class C and D inherit from class B. Later we find that class A (base class of B) has a bug that must be corrected. After correcting the bug in A, it automatically will be reflected across all classes that inherit from A, if the class A has been inherited without changes. See the reduction in amount of efforts that one would have done if each class inherited from A was to be modified separately, a gifted benefit of being transitive.

Some Facts and Terms. When we say that the class 'Student' inherits from the class 'Person' then 'Person' is a *base class* of 'Student' and 'Student' is a *subclass* (or derived class) of 'Person'.

Although 'Student' is a 'Person' yet the reverse is not true. A 'Person' need not be 'student'. All members of 'Student' are not members of class 'Person'. The class 'Student' has properties it does not share with class 'Person'. For instance, 'Student' has a 'marks-percentage', but 'Person' does not have.

Note A subclass defines only those features that are unique to it.

3.3.5 Polymorphism

Polymorphism is a key to the power of object-oriented programming. It is so important that languages that don't support polymorphism cannot advertise themselves as OO languages. Languages that support classes but not polymorphism are called object-based languages. Ada is such a language.

Polymorphism is the concept that supports the capability of an object of a class to behave differently in response to a message or action. For instance, 'Human' is a subclass of 'Mammal'. Similarly 'Dog', 'Cat', are also subclasses of 'Mammal'. Mammals can see through day-light. So if a message 'see through daylight' is passed to all mammals, they all will behave alike. Now if a message 'see through night' is passed to all mammals, then humans and dogs will not be able to view at night whereas cats will be able to view during night also. Here cats (mammals) can behave differently than other mammals in response to a message or action. This is polymorphism.

Polymorphism is the ability for a message or data to be processed in more than one form.

Take another example. If you give $5 + 7$, it results into 12, the sum of 5 and 7. And if you give ' A ' + ' BC ', it results into ' ABC ', the concatenated strings. The same operation symbol '+' is able to distinguish between the two operations (summation and concatenation) depending upon the data type it is working on.

Here we can define polymorphism in context of OO programming also as follows :

Polymorphism is a property by which the same message can be sent to objects of several different classes, and each object can respond in a different way depending on its class.

Adjacent Fig. 3.3 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.

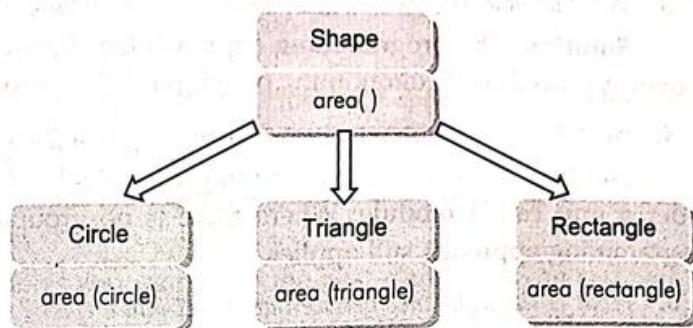


Figure 3.3 Polymorphism.

Let Us Revise

- ❖ Low Level Languages (LLs) i.e., machine language and assembly language are more close to the machine as these are machine oriented and require extensive knowledge of computer circuitry.
- ❖ High Level Languages (HLLs) are more close to the programmer as these offer English like keywords and programming constructs (sequence, selection, iteration).
- ❖ Procedural Programming aims more at procedures. The emphasis is on doing things. Data takes the back seat here.
- ❖ Modular Programming combines related procedures in a module and hides data under modules. The data are dependent on the functions. The arrangement of data can't be changed without modifying all the functions that access it.
- ❖ Object oriented programming is based on the principles of data hiding, abstraction, encapsulation, modularity, inheritance, and polymorphism. It implements programs using the objects in an object-oriented language.
- ❖ Object represents data and its associated functions under single unit.
- ❖ Class represents a group of similar objects.
- ❖ Abstraction is the way of representing essential features with background details.
- ❖ Encapsulation is the way of combining data and its associated functions under single unit. Encapsulation implements abstraction.
- ❖ Modularity is the property of decomposing a system into a set of cohesive and loosely coupled modules.
- ❖ Inheritance is the capability of a class to inherit properties from another class. That class that inherits from other class is subclass or derived class and the other class is base class.
- ❖ Inheritance supports reusability and is transitive in nature.
- ❖ A subclass defines only those features that are unique to it, rest it inherits from its base class.
- ❖ Polymorphism is the ability for a message or data to be processed in more than one form. The same operation is performed differently depending upon the data type it is working upon.

Solved Problems

1. Write a short note on the programming in low level languages.

Solution. Low level languages (machine language and assembly language) are machine-oriented and require extensive knowledge of machine circuitry. In machine language, the instructions are given in binary codes whereas in assembly language, instructions are given in symbolic names (mnemonics).

2. Write a short note on the programming in high level languages.

Solution. High Level Languages (HLLs) like BASIC, PASCAL etc. offer English like keywords, programming constructs for sequence, selection (decision) and iteration (looping), and allow use of variables and constants. Programmers feel much at ease while working in HLLs as compared to low level languages.

3. What do you understand by procedural programming paradigm ?

Solution. The programming approach that focuses on the procedures for the solution of a problem is known as procedural programming paradigm. This approach emphasizes on the 'doing' rather than the 'data'.

4. How is modular programming approach different from procedural programming approach ?

Solution. In modular programming, a set of related procedures with the data they manipulate is combined under a unit called module. Where there is no grouping of procedures with related data, the procedural programming approach still applies.

5. What is the difference between an object and a class ?

Solution. An object is an identifiable entity with some characteristics and behaviour. It represents an entity that can store data and its associated functions.

A class is a group of objects that share common properties and relationships. It represents a group of similar objects.

6. What do you mean by Abstraction and Encapsulation ? How are these two terms interrelated ?

Solution. Abstraction is the act of representing essential features without including the background details.

Encapsulation is the way of combining both data and the functions that operate on the data under a single unit.

Encapsulation is the way of implementing abstraction.

7. What is base class ? What is derived class ? How are these two interrelated ?

Solution. A derived class is a class that inherits properties from some other class.

A base class is a class whose properties are inherited by derived class.

A derived class has nearly all the properties of base class but the reverse of it is not true.

8. Explain the transitive nature of inheritance.

Solution. The transitive nature of inheritance states that if a class A inherits properties from its base class B then all its subclasses will also be inheriting the properties of base class of A i.e., B.

9. What is the benefit of transitive nature of inheritance ?

Solution. Suppose a class A inherits from class B. Classes C and D inherit from A. If there is a bug in B then correction is required only in B as it will be automatically reflected to all subclasses A, C and D if class B has been inherited without changes.

10. What is polymorphism ?

Solution. It is ability for a message or data to be processed in more than one form. It is a property by which the several different objects respond in a different way (depending upon its class) to the same message.



Glossary

Abstraction The act of representing essential features without including the background details or explanations.

Base / Super Class A class whose properties are inherited by other classes (its subclasses).

Class Group of objects that share common properties and relationships.

Derived / Sub Class A class which inherits properties from its base class.

Encapsulation Wrapping up of data and functions (that operate on the data) into a single unit.

Inheritance Capability of one class of things to inherit capabilities or properties from another class.

Modularity The property of decomposing a system into a set of cohesive and loosely coupled modules.

Modular Programming Paradigm Decide which modules you want ; partition the program so that data is hidden in modules.

Module A set of related procedures with the data they manipulate.

Object Identifiable entity with some characteristics and behaviours.

Object Oriented Paradigm Decide which classes and objects are needed ; provide a full set of operations for each class.

Paradigm Organizing principle of a program. An approach to programming.

Polymorphism Property by which the same message can be sent to objects of several different classes, and each object can respond in a different way depending on its class.

Procedural Programming Paradigm Decide which procedures you want ; use the best algorithms you can find.

Subclass Derived class.

A ssignments

TYPE A : VERY SHORT ANSWER QUESTIONS

1. What are the two major types of programming languages ?
2. Which two programming languages are low level languages ?
3. How are programs written in (i) machine language, (ii) assembly language ?
4. Why are low level languages considered close to the machine ?
5. Why is it easier to program with high level languages ? Why are high level languages considered close to the programmer ?
6. Which two related purposes should be served by a programming language ?
7. Why is 'C++' called 'middle level language' ?
8. What do you understand by programming paradigm ? Name various programming paradigm.
9. What are the characteristics of procedural paradigm ?
10. What is a module ? What is modular programming paradigm ? What are its characteristics ?
11. In procedural programming, the emphasis is on doing things. True or False ?
12. In procedural programming, data is the reason for a program's existence. True or False ?
13. In procedural programming, the program revolves around the data. True or False ?
14. In modular programming, procedural approach is used when procedures are not grouped in a module. True or False ?
15. Modular programming allows the modification of data without affecting the functions associated. T/F ?
16. Real world cannot be modelled using procedural or modular approach. True or False ?
17. Procedural programming makes the software reuse possible. True or False ?
18. The object-oriented approach views a problem in terms of objects involved. True or False ?
19. Procedures take a secondary status in object-oriented approach. True or False ?
20. What is an object ? What is a class ? How is an object different from a class ?
21. What is object-oriented programming paradigm ? Name the four basic concepts of OOP ?
22. What is meant by Abstraction ?
23. The real world concept gets simplified using concept of abstraction. True or False ?

24. Give an example to illustrate the concept of abstraction.
25. What is encapsulation ? Why is data considered safe if encapsulated ?
26. How are the terms abstraction and encapsulation related ?
27. Why are classes called Abstract Data Type ?
28. What is a baseclass ? What is a subclass ? What is the relationship between a baseclass and subclass ?
29. What is modularity ? What benefits does it offer ?
30. Inheritance ensures closeness with the real-world models. True or False ?
31. How does inheritance support 'reusability' ?
32. What do you mean by the transitive nature of inheritance ?
33. A subclass defines all the features of baseclass and its additional features. True or False ?
34. A subclass inherits all the properties of its baseclass and vice versa. True or False ?
35. What are object-based languages ? Give an example of object-based language.
36. What is polymorphism ? Give an example illustrating polymorphism.

TYPE B : SHORT ANSWER QUESTIONS

1. Briefly explain the two major types of programming languages.
2. Write a short note on programming in two major types of languages.
3. How are high level languages different from low level languages ?
4. What do you mean by 'middle level languages' ? Why are these called middle-level languages ? Give examples of such languages.
5. What are programming paradigms ? Give names of some popular programming paradigms.
6. Write a short note on procedural programming.
7. Write a short note on modular programming.
8. What are the shortcomings of procedural and modular programming approaches ?
9. Write a short note on OO programming.
10. How does OOP overcome the shortcomings of traditional programming approaches ?
11. What is the difference between object and class ? Explain with examples.
12. Explain briefly the concept of data abstraction with the help of an example.
13. Explain briefly the concept of encapsulation with the help of an example.
14. How the data is hidden and safe if encapsulation is implemented ? Explain with example.
15. Simulate a daily life example (other than the one mentioned in the chapter) that explains encapsulation.
16. Write a short note on inheritance.
17. What are the advantages offered by inheritance ?
18. How is reusability supported by inheritance ? Explain with example.
19. What is the benefit of transitive nature of inheritance ? Explain with example.
20. What is polymorphism ? Explain with example.
21. Do you think OOP is more closer to real world problems ? Why ? How ?

TYPE C : LONG ANSWER QUESTIONS

1. Write a note on software evolution.
2. Explain different programming paradigms, their shortcomings etc. with proper examples.
3. Explain the basic concepts of OOP with examples.

Introducing Java

4.1 Introduction

Java is a popular third-generation programming language, which can be used to do any of the thousands of things that a computer software can do. With the features it offers, Java has become the language of choice for Internet and Intranet applications. Java plays an important role for the proper functioning of many software-based devices attached to a network. The kind of functionality the Java offers, has contributed a lot towards the popularity of Java.

This chapter is going to talk about a brief history of Java, its important features, functionality etc. In this chapter, you shall also learn about how to execute java programs.

4.2 About Java

Java is both a *programming language* and a *platform*. Like any other programming language, you can use Java to write or create various types of computer applications. Thus, Java fits the definition of a programming language. Java is also a *platform* for application development. The word *platform* generally is used to refer to some *combination of hardware and system software e.g., operating system Windows Vista on Intel Pentium V or Windows NT on DEC Alphas or System 8.5 on PowerMacs etc.*

In This Chapter

- 4.1 Introduction
- 4.2 About Java
- 4.3 Simple Java Program
- 4.4 Creating and Running a JAVA Program

The Java Platform is a new software platform different from many other platforms ; it is designed to deliver and run highly interactive, dynamic and secure applications on networked computer systems. We are not going into further details of *Java platform* as it is beyond the scope of this book.

4.2.1 History of Java

Do you know that Java was not developed keeping in mind the World Wide Web ? Originally, Java started as an elite project (code named **Green**) to find a way of allowing different electronic devices such as TV-top boxes and controllers to use a common language. This language for electronic devices was originally named **Oak** but failed to find a niche despite its potential.

Java Programming Language was written by *James Gosling* along with two other persons *Mike Sheridan* and *Patrick Naughton*, while they were working at Sun Microsystems. Ever since its introduction, it has gone through many changes.

Major milestones in Java history are: *release of source code of Java virtual machine (JVM) as free and open-source software, (FOSS), under the terms of the GNU General Public License (GPL) in the first decade of 21st century; and acquisition of Sun Microsystems by Oracle Corporation by the end of the same decade.*

Java comes in various platforms :

- ❖ **Java Card** for smartcards.
- ❖ **Java Platform, Micro Edition (Java ME)** – targeting environments with limited resources.
- ❖ **Java Platform, Standard Edition (Java SE)** – targeting workstation environments.
- ❖ **Java Platform, Enterprise Edition (Java EE)** – targeting large distributed enterprise or Internet environments.

4.2.2 Byte Code

Do you know that computer programs are very closely tied to the specific hardware and operating system they run on. For example, a *Windows program* will not be able to run on a computer that only runs *DOS* ; A *Mac application* can't run on a *Unix workstation*, and so on. To tackle with diversity of platforms, major commercial applications like *MS-Word* or *Netscape Navigator* have to be written almost independently for all different platforms they run on.

But for the applications developed with Java, this is not the case. For, these applications are platform-independent i.e., they are not affected with changing platforms. Java solves the problem of *platform-independence* by using **byte code**. In order to understand the byte code, you must be clear about compilation process first, which is being discussed below.

Ordinary Compilation Process

As you know that a program is a set of instructions given to a computer. The program or code written by a programmer is usually called the **Source Code**. This source code needs to be converted into *machine language code*, which a computer can easily understand. The process of converting a source code into machine code, is called **compilation**. The converted machine

code depends a lot on the platform it is executing upon. That means for different platforms different machine code is produced [Fig. 4.1(a)]. This resultant machine code is called *native executable code*.

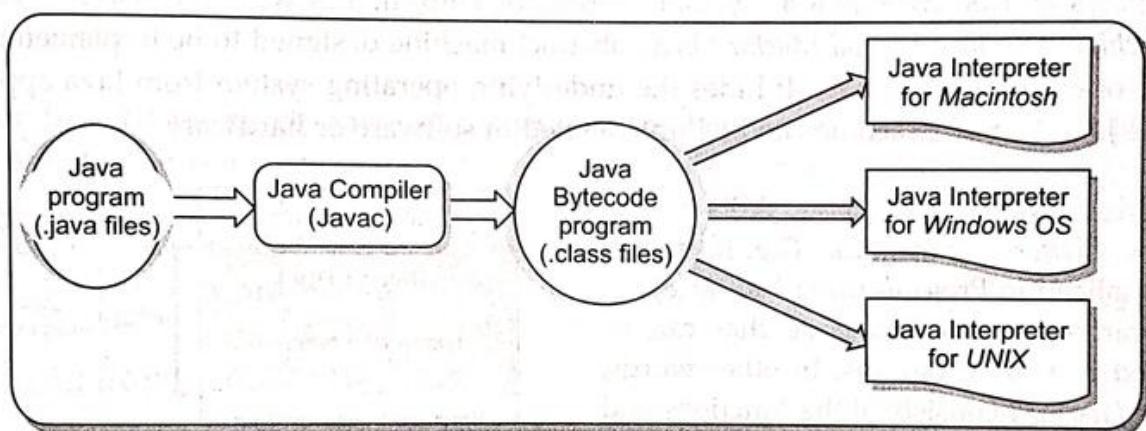
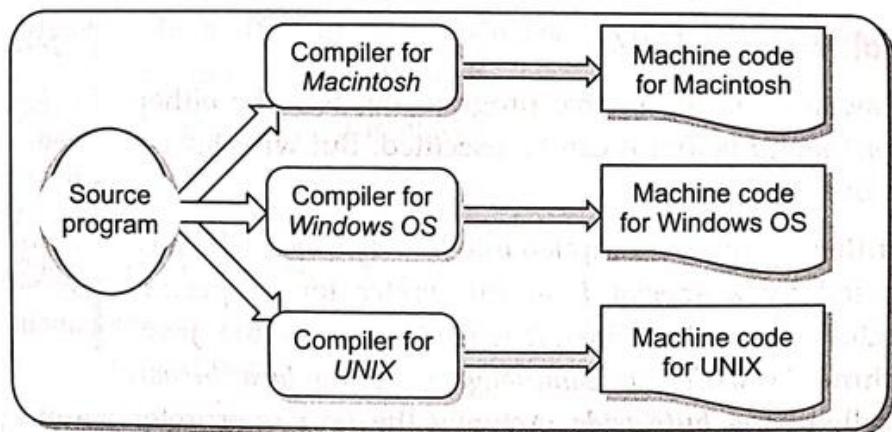


Figure 4.1 (a) Ordinary Compilation Process. (b) Java Compilation producing bytecodes.

Java compiler is **javac** – a program that compiles Java source code (Java programs) into compiled byte code.

There is an alternative to compiling a high-level language program. Instead of using a *compiler*, which translates the program **all at once**, you can use an *interpreter*, which translates it **instruction-by-instruction**, as necessary. In order to execute a program, the interpreter repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so.

Java Compilation

Contrary to ordinary compilers, the Java compiler does not produce *native executable code* for a particular machine. Instead it produces a special format called *byte code*.

The **Java byte code** looks a lot like machine language, but unlike machine language **Java byte code is exactly the same on every platform**. This byte code means the same thing on a *Solaris workstation* as it does on a *Macintosh Power Book* or on *Windows Vista* running on an *Intel Pentium V*. However, the Java programs that have been compiled into byte code still need an *interpreter* to execute them on any given platform [see Fig. 4.1(b)]. The interpreter reads the byte code and translates it into the *native*

The Java Byte Code is a machine instruction for a Java processor chip called *Java Virtual Machine*. The byte code is independent of the computer system it has to run upon,

language of the host machine on the fly. Since the byte code is completely platform independent, only the interpreter and a few native libraries need to be ported to get Java to run on a new computer or operating system.

4.2.3 Java Virtual Machine (JVM)

As you are aware that any source program needs to be either *compiled* or *interpreted* before it can be executed. But with Java a combination of these two is used.

Programs written in Java are compiled into *Java Byte code*, which is then interpreted by a special *Java Interpreter* for a specific platform. Actually this *Java interpreter* is known as the **Java Virtual Machine (JVM)**. The *machine language* for the *Java Virtual Machine* is called *Java byte code*. Actually the Java interpreter running on any machine appears and behaves like a "virtual" processor chip, that is why, the name – *Java Virtual Machine*. The *Java Virtual Machine* is an abstract machine designed to be implemented on the top of existing processors. It hides the underlying operating system from Java applications. The Java Virtual Machine can be implemented in software or hardware.

JVM combined with **Java APIs** makes **Java Platform** (Fig. 4.2). The **Java API** (Application Programming Interface) are libraries of compiled code that can be used in your programs. In other words, the *Java API* consists of the functions and variables that programmers are allowed to use in their applications.

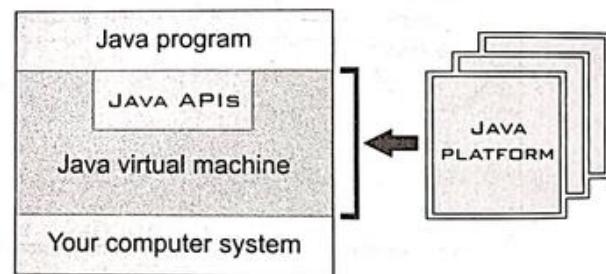


Figure 4.2 The Java Platform.

Thus, you can write a Java program (.java files) on any platform and use the JVM compiler (called **javac**) to generate a bytecode file (.class files). This bytecode file can be used on any platform that has installed Java. However, **do note that bytecode is not an executable file**. To execute a bytecode file, you actually need to invoke a *Java interpreter* (called **java**). Every platform has its own Java interpreter which will automatically address the platform-specific issues. When platform-specific operations are required by the bytecode, the Java interpreter fills this need by linking in appropriate code specific to the platform.

The JVM compiler (**javac**) compiles the Java source code/program file (.java files) into byte code (.class files), which can then be executed using Java interpreter program (**java**) (Explained in section 4.4)

4.2.4 Characteristics of Java

Although Java has many and many characteristics that make it eligible for a powerful and popular language. In the following lines, we are going to discuss a few important characteristics of Java.

- ◆ **Write Once Run Anywhere (WORA).** The Java programs need to be written just once, which can be run on different platforms without making changes in the Java program. Only the Java interpreter is changed depending upon the platform.

Note Java programs are compiled and their byte codes are produced. The byte codes are always exactly the same irrespective of the computer system they are to execute upon.

- ❖ **Light Weight Code.** With Java, big and useful applications can also be created with very light code. No huge coding is required.
- ❖ **Security.** Java offers many security features to make its programs safe and secure.
- ❖ **Built-in Graphics.** Java offers many built-in graphics features and routines which can be used to make Java application more visual.
- ❖ **Object-Oriented Language.** Java is object-oriented language, thereby, very near to real world.
- ❖ **Supports Multimedia.** Java is ideally suited for integration of video, audio, animation and graphics in Internet environment.
- ❖ **Platform Independent.** Java is essentially platform independent. Change of platform does not affect the original Java program/application.
- ❖ **Open Product.** Java is an open product, freely available to all. However, there exist some special time-saving Java development kits, which can be availed by paying small amounts.

After this, let us move on to the discussion of how to create programs in BlueJ Java environment. It is not necessary to have BlueJ always to create and run Java programs. You can also create Java program in a simple text editor like Notepad and compile & run it by issuing few commands. This way of creating and running Java programs has been given in Appendix A. But before that you must be aware of various components of a Java program.

4.3 Simple Java Program

Let us now have a look at an example Java program. Following lines show a simple and short example-java-program – **HelloWorld.java**.

```
/* program HelloWorld.java */
class HelloWorld
{
    public static void main(String args[ ])
    {
        System.out.println("Hello World!!") ;
    }
} // main ends here
```

Let us now examine parts of this sample program.

- **Initial class *HelloWorld***

The line *class HelloWorld* means that a class is being defined.

Notice that the classname is **HelloWorld** in the program. Also this class is *initial class*. An *initial class* is the one that defines the program name. (Note that the program name is **HelloWorld.java** i.e., same name as that of *initial class*.) An *initial class* contains *main* function inside it. Although there can be many classes in a Java program but there can be only one initial class in a Java program.

■ **Method main**

The *HelloWorld* class contains one method – the **main** method. The **main** method is where an application begins executing. If there are multiple classes in a program, the execution will begin from the initial class *i.e.*, the class containing **main** method. If no **main** method is there, no execution will take place. Thus **main** method is also termed, sometimes, the driver method – it drives your program.

See, the **main** method is declared as :

```
public static void main .....
```

The significance and meanings of keywords **public**, **static**, **void** will be discussed later as you won't be able to understand it at this very stage.

■ **System.out.println("Hello World!!")**

This statement prints *Hello World!!* on the standard output device which is generally your monitor.

■ **Comments /*.....*/ and // ...**

The text enclosed within **/*.....*/**, called **comments**, is purely for enhancing readability.

The comments are ignored by the compiler and not executed at all even if you write a valid Java statement within **/*.....*/**. Within **/*.....*/**, multi-line comments can be inserted but with **//....** only single line comment can be inserted.

Figure 4.3 highlights various parts of a Java program.

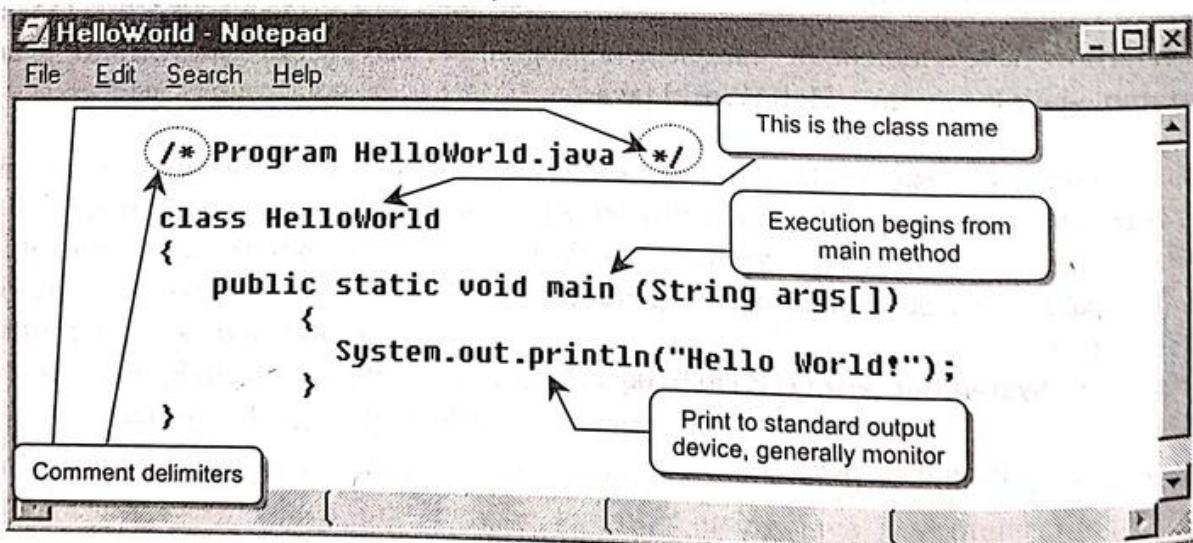


Figure 4.3 Parts of a simple Java program.

4.3.1 Types of Java Programs

There are *two* types of Java programs

- ❖ Internet applets and
- ❖ Stand alone applications.

The *first* type of Java program, **Internet Applets** are small programs that are embedded in web pages and are run on the viewers machine in a secured manner (*which means limited access to system resources i.e., the hard disk*) by Java capable browsers.

Applets are designed to be delivered to Internet Web browsers and that is why an applet has a built-in graphical window. But Java applets have some security restrictions (e.g., it cannot write to a local file).

The *second* type of Java program, **stand alone application**, is much more interesting. It is generally a software application that does not require low level operating system or hardware access. This includes most of the usual desktop applications such as word processors or spreadsheets. Stand alone Java applications could be distributed on standard ISO 9660 format CD-ROMs and installed on any Java capable machine. Every stand alone application of Java begins executing with a *main* method.

After learning about various parts of a Java program, let us now learn how to create and execute a Java program.

4.4 Creating and Running a Java Program

Let us now learn how you can create and run Java program. We shall be talking about two methods of creating and running Java programs. These are :

- (a) General Method
- (b) Using JCreator IDE

4.4.1 General Method of Creating and Running Java Programs

The method that we are discussing here can work on different operating systems (such as Windows 95, Windows 98, Windows XP), that provide an MS-DOS prompt. Under this method, to create and run Java programs, you need to follow these steps.

1. Create a Java Source File

- (i) Start an editor such as WordPad or Notepad. In a new document, type in the desired Java source code.
- (ii) Save this code to a file having same name as that of initial class and extension .java. e.g., the class that contains method main has name as *HelloWorldApp*, then the code should be saved as *HelloWorldApp.java*

2. Compile the Source File

From the Start menu, select the **MS-DOS Prompt** application (Windows 95/98) or **Command Prompt** application (Windows NT). If you have Windows XP installed, then you need to click at **Start → Run** and then type *cmd.exe* in the *Run dialog*, to bring up the MS-DOS command prompt. When the application launches, it should look like this :

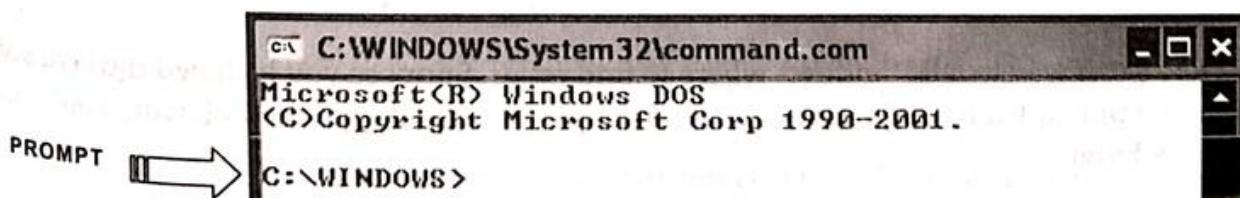


Figure 4.4 (a) A DOS Prompt.

The prompt shows your *current directory*. When you bring up the prompt for Windows 95/98, your current directory is usually WINDOWS on your C drive (as shown above) or WINNT for

Windows NT. To compile your source code file, change your current directory to the directory where your file is located. For example, if your source directory is *java* on the *C* drive, you would type the following command (*cd¹* command) at the prompt and press **Enter**:

cd c:\java

Now the prompt should change to *C:\java>*.

As shown here, to change to the *java* directory on the *D* drive, you must reenter the drive, *d*:

If you type *dir²* at the prompt, you should see all the files under current directory including your file, *HelloWorldApp.java*.

Note To change to a directory on a different drive, you must type an extra command.

```
C:\WINDOWS> CD D:\JAVA
C:\WINDOWS>D:
D:\JAVA>
```

Figure 4.4 (b) Changing Directory through CD command.

```
C:\WINDOWS\System32\command.com
C:\JAVA>DIR
Volume in drive C has no label.
Volume Serial Number is 98A4-F2D2

Directory of C:\JAVA
02/02/2016  02:06 PM    <DIR>
02/02/2016  02:06 PM    <DIR>
02/17/2018  03:05 PM           171 HelloWorld.java
                  1 File(s)      171 bytes
                  2 Dir(s)  17,863,704,576 bytes free

C:\JAVA>
```

Figure 4.4 (c) Viewing list of files through DIR command.

Now you can compile. At the prompt, type the following command and press **Enter**:

javac HelloWorld.java

If your prompt reappears without error messages, *congratulations*. You have successfully compiled your program. However, if you see any type of error messages then carefully read the following lines.

Error Explanation

Bad command or file name

If you receive above shown error, this means Windows could not find the Java compiler, *javac*.

Here's one way to tell Windows where to find *javac*. Suppose you installed the Java Software Development Kit in *C:\jdk9.0.4*. At the prompt you would type the following command and press **Enter**:

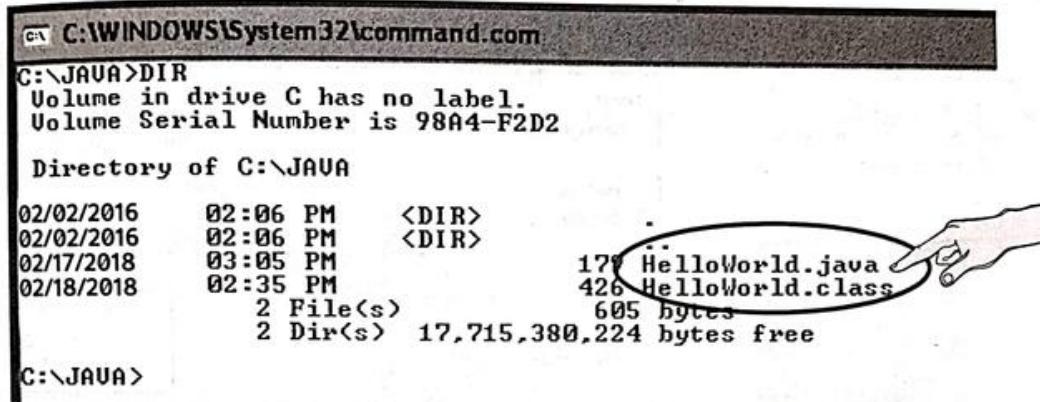
Path = %path%; C:\jdk9.0.4\bin

1. CD is a DOS command, used for changing current working directory
2. Dir is a DOS command that lists names of files under one directory.

And then issue earlier command once again i.e.,

```
javac HelloWorld.java
```

After successful compilation, the compiler has generated a Java bytecode file, *HelloWorld.class*. At the prompt, type dir to see the new file that was generated :



```
C:\WINDOWS\System32\command.com
C:\JAVA>DIR
Volume in drive C has no label.
Volume Serial Number is 98A4-F2D2

Directory of C:\JAVA

02/02/2016  02:06 PM    <DIR>
02/02/2016  02:06 PM    <DIR>
02/17/2018   03:05 PM
02/18/2018   02:35 PM      179 HelloWorld.java
                  426 HelloWorld.class
                    2 File(s)       605 bytes
                    2 Dir(s)  17,715,380,224 bytes free

C:\JAVA>
```

Figure 4.4 (d) The class file comes after successful compilation.

Now that you have a .class file, you can run your program as explained in the following step.

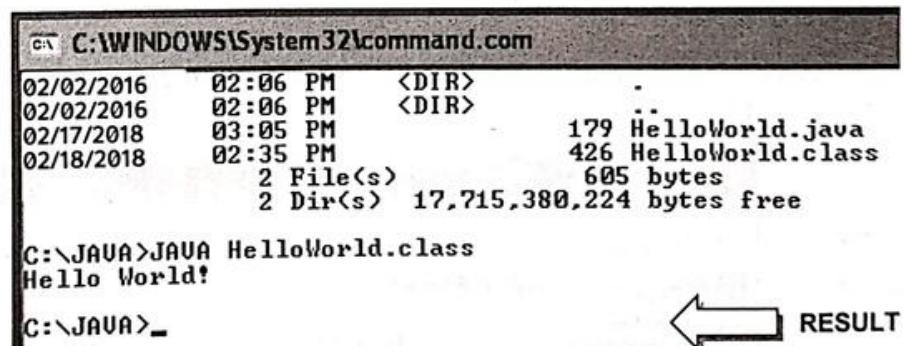
3. Run the Program

In the same directory, now type the following command at the prompt and press Enter.

```
java HelloWorldApp
```

It will now show you the result of your program.

Congratulations!
Your program works.



```
C:\WINDOWS\System32\command.com
02/02/2016  02:06 PM    <DIR>
02/02/2016  02:06 PM    <DIR>
02/17/2018   03:05 PM
02/18/2018   02:35 PM      179 HelloWorld.java
                  426 HelloWorld.class
                    2 File(s)       605 bytes
                    2 Dir(s)  17,715,380,224 bytes free

C:\JAVA>JAVA HelloWorld.class
Hello World!
C:\JAVA>_
```

RESULT

Figure 4.4 (e) Running Java Program

4.4.2 Creating and Running Java Program Using JCreator LE

You can also create and run Java programs in many softwares. Out of these JCreator is my personal favourite because of these reasons :

- ❖ its light weight version (JCreator LE) is a freeware (which can be downloaded from [www.jcreator.com/download.htm³](http://www.jcreator.com/download.htm)),
- ❖ it is an IDE (Integrated Development Environment) i.e., editor, compiler, executor etc are all available together, and
- ❖ its look and feel is like professional software.

³ Make sure to download JCreator LE, which is a freeware. Do not download JCreator Pro, it has to be purchased.

Before you install JCreator, make sure that Java (JDK1.X) is installed on your computer. Once it is there, you can create and run Java programs in *JCreator LE* as explained below :

1. Start JCreator LE

Double click on JCreator's shortcut on the desktop. It will open JCreator IDE⁴ for you.

Once in JCreator, click on File → New → File (see below) or press Ctrl+N.

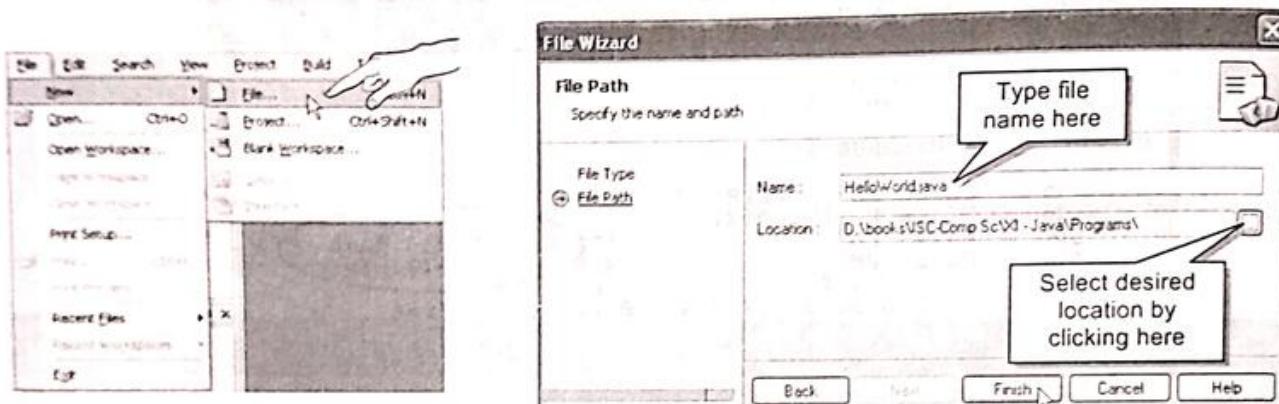


Figure 4.5 (a)

(b)

2. Specify file name and location

It will bring up *File Wizard* wherein type the filename and directory where it is to be stored. We specified *HelloWorld.java* (see Fig. 4.5(b)). And then click **Finish**.

3. Type the Code

Now it will create workspace for you. Start typing your program code in the right pane [see Fig. 4.5(c)]

The image shows the JCreator workspace for the file 'HelloWorld.java'. The code area contains:

```

1 * Program HelloWorld.java */
2 class HelloWorld {
3
4     public static void main (String args[])
5     {
6         System.out.println("Hello World!");
7     }
}

```

A callout box points to the code area with the text 'Type your program code in this pane'. Another callout box points to the status bar at the bottom with the text 'At compilation errors will be shown in this pane'.

Figure 4.5 (c)

4. On First run it may ask to specify default file associations and Project directory. We selected .java as type of file to be associated.

4. Compile and Execute

After typing the program, firstly compile it by clicking at **Build → Compile File** command. In case of errors, the errors' list will be shown in the bottommost pane of IDE window [see Fig. 4.5(c)]. You'll need to rectify the errors and recompile the code. And once error free, run the compiled code, click at **Build → Execute file** command. It will show you the result in a separate output window [see Fig. 4.5(d)].

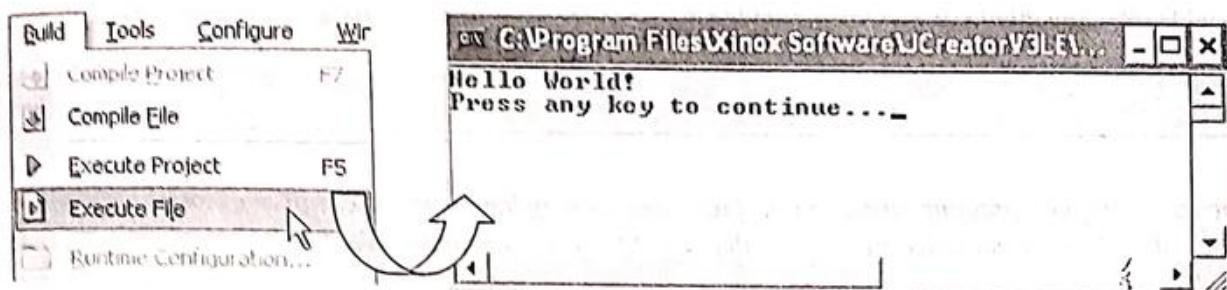


Figure 4.5 (d)

Types of Program Errors

Before you run your program, you need to compile. The compiler is capable of finding some types errors and once your program is reported error-free by the compiler, you can execute it. "No compile time error" does not guarantee the correct and proper execution of your program. Some errors may occur while executing your program, too.

Broadly the types of errors based on what caused them, can be :

- Syntax errors** errors due to the fact that the syntax of the language is not respected.
- Semantic errors** errors due to an improper use of program statements.
- Logical errors** errors due to the fact that the specification is not respected.

From the point of view of when errors are detected, errors can be these types :

- Compile time errors** syntax errors and static semantic errors indicated by the compiler.
- Runtime errors** dynamic semantic errors, and logical errors, that cannot be detected by the compiler (debugging).

In order to understand these types of errors, it is important that you understand them with some code examples. Thus we shall get back to this discussion of errors and a related concept of Exception in a later chapter – **Program Error Types and Basic Exception Handling in Java** after you have learnt about some basic code of Java.

Well, here we are through with the chapter. Let us quickly brush up what we have learnt in this chapter.

Let Us Revise

- ❖ Java is a popular third generation programming language.
- ❖ Java was originally named as Oak.
- ❖ Java byte code is a machine instruction for a Java processor chip called Java virtual machine.

- ❖ The Java byte code is independent of the computer system it has to run upon.
- ❖ The byte codes are always exactly the same irrespective of the computer system they are to execute upon.
- ❖ Java programs are compiled into Java bytecode.
- ❖ Java virtual machine is an abstract machine which is implemented on the top of existing processors.
- ❖ In a Java program there can be many classes but only one initial class.
- ❖ There are two types of Java applications-Internet applet and stand alone application.
- ❖ An Internet applet runs within a web browser.
- ❖ A standalone application can run on any platform.

Solved Problems

- When you compile a program written in the Java programming language, the compiler converts the human readable source file into platform independent code that a JVM can understand ? What is this platform independent code called ?

Solution. Byte code

- How can you say that Java is both a programming language and a platform ?

Solution. Like any other programming language, we can use Java to write or create various types of computer applications. The word *platform* generally is used to refer to some combination of hardware and system software. The Java platform is a new software platform designed to deliver and run highly interactive, dynamic and secure applications on networked computer systems.

- How is ordinary compilation process different from Java compilation ?

Solution. In ordinary compilation, the source code is converted to a machine code, which is dependent upon the machine or the platform. This resultant machine code is called *native executable code*.

Contrary to ordinary compilers, the Java compiler does not produce *native executable code* for a particular machine. Instead it produces a special format called *byte code*. The Java *byte code* looks a lot like machine language, but unlike machine language *Java byte code is exactly the same on every platform*.

- What do you understand by JVM ?

Solution. The *Java Virtual Machine* is an abstract machine designed to be implemented on the top of existing processors. It hides the underlying operating system from Java applications. Programs written in Java are compiled into *Java Byte code*, which is then interpreted by a special *Java Interpreter* for a specific platform. Actually this *Java interpreter* is known as the *Java Virtual Machine (JVM)*.

- What is Write Once Run Anywhere characteristic of Java ?

Solution. The Java programs need to be written just once, which can be run on different platforms without making changes in the Java program. Only the Java interpreter is changed depending upon the platform.

This characteristic is known as Write Once Run Anywhere.

- What will be the result produced by following Java program ?

```
class MyClass{
    public static void main (String [ ] args) {
        System.out.println ("Printed from inside of");
        System.out.println ("MyClass");
    }
}
```

Solution. Printed from inside of
MyClass

7. Change the program of question 6, so that it prints :

Printed from inside of MyClassRoom
Thank You.

Solution.

```
class Myclass{
    public static void main (String [ ] args) {
        System.out.println ("Printed from inside of MyclassRoom") ;
        System.out.println ("Thank You") ;
    }
}
```

8. What is an initial class ?

Solution. A Java program can contain many classes. But one class in a Java program can contain the main method. This class is called initial class.

9. What is the significance of method main ?

Solution. All Java applications begin their execution from *main* method. If no *main* method is there in a Java program, then no execution takes place.

10. Name two types of Java programs ?

Solution. The two types of Java applications are *Internet applets* and *Stand alone applications*.

11. How are Internet applets different from stand alone applications of Java ?

Solution. Internet applets cannot run on their own. They run from within a web browser whereas Java stand alone applications can run independently on any platform.

12. What commands will be typed on command prompt to compile and run a program stored in file *MyProg.java*.

Solution.

```
Javac.MyProg.java
Java MyProg
```

Glossary

Bytecode A machine instruction for a Java processor chip called Java Virtual Machine (JVM).

Java Virtual Machine (JVM) An abstract machine that hides underlying operating system from Java applications and executes the Java byte code.

Java API Java Application Programming Interface. Libraries of compiled code that can be used in Java programs.

Assignments

TYPE A : VERY SHORT/SHORT ANSWER QUESTIONS

1. Why is Java often termed as a platform ?
2. What is bytecode ?
3. How is Java platform independent ?

4. What is JVM ?
5. Describe ordinary compilation process ?
6. Describe Java compilation.
7. How is Java bytecode executed ?
8. What are Java APIs ?
9. Write a simple Java program that prints your name. Compile and run it.
10. Add comments to your previous program. Once again compile and run it.

TYPE B : LONG ANSWER QUESTIONS

1. Java is both a programming language and a platform. Comment.
2. Briefly write the history of Java.
3. How are bytecode and platform independence interlinked ?
4. What is the role of JVM in platform independence ?
5. What are different characteristics of Java ?
6. Create a Java program namely *JProg.java* that prints
JAVA IS WONDERFUL.
7. Now compile and run the program you created just now.
8. Add another class (given below) in the same program i.e. in *JProg.java*

```
public class Two {
    public static void display( ) {
        System.out.println ("Hi There !!") ;
        System.out.println ("I am in class Two") ;
    }
}
```

9. Now add following code in method main() of *JProg.java*, below

```
System.out.println("JAVA IS WONDERFUL") ;
display( );
```

Recompile and run your program. See what happens.

10. Now create another program with following class in it :

```
class Three {
    public static void check( ) {
        System.out.println ("I am in class three") ;
    }
}
```

Compile and run it. See what happens ? (It is linked to inavailability of main method).

Java Fundamentals

In This Chapter

-
- 5.1 Introduction
 - 5.2 Java Character Set
 - 5.3 Tokens
 - 5.4 Concept of Data Types
 - 5.5 Variables
 - 5.6 Constants
 - 5.7 Operators in Java
 - 5.8 Expressions
 - 5.9 Java Statements
 - 5.10 Significance of Classes
 - 5.11 Objects as Instances of Class

5.1 Introduction

In any language, there are some fundamentals that you need to know before you can write even the most elementary programs. This chapter introduces Java fundamentals to you so that you may start writing your own Java programs. You will be learning about Java tokens such as keywords, identifiers, literals, operators and separators, along with data types, variables, constants, operators, expressions. You'll also be learning about how a class forms the basis of all computation in Java. Let us begin our discussion with Java characters set and tokens.

5.2 Java Character Set

Character set is a set of valid characters that a language can recognise. A character represents any letter, digit or any other sign.

Java uses the **Unicode** character set. **Unicode** is a two-byte character code set that has characters representing almost all characters in almost all human alphabets and writing systems around the world including English, Arabic, Chinese and many more.

The first 128 characters in the *Unicode* character set are identical to the common *ASCII character set*. The second 128 characters are identical to the upper 128 characters of the *ISO Latin-1 extended ASCII character set*. It's the next 65,280 characters that present problems.

You can refer to a particular *Unicode* character by using the escape sequence \u followed by a four digit hexadecimal number.

For example,

\u00AE	©	The copyright symbol
\u0022	"	The double quote
\u00BD	½	The fraction 1/2
\u0394	Δ	The capital Greek letter delta

You can even use the full *Unicode* character sequence to name your variables. However chances are your text editor won't be able to handle more than basic ASCII very well.

5.3 Tokens

In a passage of text, individual words and punctuation marks are called tokens. In fact every unit that makes a sentence is a token.

Java has the following tokens :

- | | | |
|----------------|----------------|-------------|
| 1. Keywords | 2. Identifiers | 3. Literals |
| 4. Punctuators | 5. Operators | |

The smallest individual unit in a program is known as a **Token**.

5.3.1 Keywords

Keywords are the words that convey a *special meaning* to the language compiler. These are reserved for special purpose and must not be used as normal identifier names.

The following character sequences, formed from ASCII letters, are reserved for use as *keywords* and cannot be used as identifiers :

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

The keywords **const** and **goto** are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these Java keywords incorrectly appear in programs.

While **true** and **false** might appear to be keywords, they are technically *boolean literals*. Similarly, while **null** might appear to be a keyword, it is technically the *null literal*. Thus **true**, **false** and **null** are not keywords but *reserved words*.

5.3.2 Identifiers

Identifiers are fundamental building blocks of a program and are used as the general terminology for the names given to different parts of the program viz. variables, objects, classes, functions, arrays etc. Identifier forming rules of Java state the following :

1. Identifiers can have alphabets, digits and underscore and dollar sign characters.
2. They must not be a *keyword* or *boolean literal* or *null literal*.
3. They must not begin with a digit.
4. They can be of any length.
5. Java is case sensitive i.e., upper-case letters and lower-case letters are treated differently.

The following are some *valid* identifiers :

Myfile	DATE9_7_77	Z2T0Z9	A_to_Z
MYFILE	_DS	_HJI3_JK	isLetterorDigit
_CHK	FILE13	αρετη	\$1_to_\$10

Note Java is case sensitive as it treats upper and lower-case characters differently.

The following are some *invalid* identifiers :

DATA-REC	contains special character - (other than A - Z, a - z and _ or \$)
29CLCT	Starting with a digit
break	reserved keyword
My.file	contains special character.

Identifier Naming Conventions

While making identifier names, certain conventions are followed. These are :

- (i) The names of public methods and instance variables should begin with a lower case letter e.g.,
`maximum sum`
- (ii) For names having multiple words, second and subsequent words beginning character is made *capital* so as to enhance readability e.g.,
`avgSalaryOfEmployees, dateOfBirth`
- (iii) Private and local variables should use lower case letters e.g.,
`width, result, final_score`
- (iv) The class names and interface names begin with an upper case letter e.g.,
`InitialClass, Employee, Student`
- (v) The constants should be named using all capital letters and underscores e.g.,
`MAX_VALUE, MAX_MARKS, SPECIAL_SALARY, TOTAL`

5.3.3 Literals

Literals (often referred to as constants) are data items that are fixed data values. Java allows several kinds of literals :

Fixed data values are
Literals.

- (i) integer-literal
- (ii) floating-literals
- (iii) boolean literals
- (iv) character-literal
- (v) string-literal
- (vi) the null literal

(i) Integer Literals

Integer literals are whole numbers without any fractional part. The method of writing integer constants has been specified in the following rule.

An integer constant must have at least one digit and must not contain any decimal point. It may contain either + or - sign. A number with no sign is assumed to be positive. Commas cannot appear in an integer constant.

Java allows *three* types of integer literals :

1. Decimal (base 10)
2. Octal (base 8)
3. Hexadecimal (base 16)

1. Decimal Integer Literals. An integer literal consisting of a sequence of digits is taken to be decimal integer constant unless it begins with 0 (digit zero). For instance, 1234, 41, +97, -17 are decimal integer literals.

2. Octal Integer Literals. A sequence of digits starting with 0 (digit zero) is taken to be an octal integer. For instance, decimal integer 8 will be written as 010 as octal integer. ($\because 8_{10} = 10_8$) and decimal integer 12 will be written as 014 as octal integer ($\because 12_{10} = 14_8$). But make sure that when an integer begins with 0, it must not contain 8 and 9 as these are invalid octal digits.

3. Hexadecimal Integer Literals. A sequence of digits preceded by 0x or 0X is taken to be an hexadecimal integer. For instance, decimal 12 will be written as 0XC as hexadecimal integer. But with Hexadecimal constants only 0-9 and A-F can be used. All other letters are illegal.

Thus number 12 will be written either as 12 (as decimal), 014 (as octal) and 0XC (as hexadecimal).

The suffix l or L and u or U attached to any constant forces it to be represented as a *long* and *unsigned* respectively.

(ii) Floating Literals

Floating literals are also called *real literals*.

Real literals are numbers having fractional parts. These may be written in one of the two forms called *fractional form* or the *exponent form*.

A real literal in *fraction form* consists of signed or unsigned digits including a decimal point between digits.

The rule for writing a real constant in fractional form is given below :

A real literal in fractional form must have at least one digit before a decimal point and at least one digit after the decimal point. It may also have either + or - sign preceding it. A real literal with no sign is assumed to be positive.

The following are valid real literals in fractional form

2.0, 17.5, -13.0, -0.00625

The following are invalid real constants

7	(No decimal point)
7.	(No digit after decimal point)
+17/2	(/-illegal symbol)
17,250.26.2	(Two decimal points)
17,250.262	(comma not allowed)

A real literal in *exponent form* consists of two parts : *mantissa* and *exponent*. For instance 5.8 can be written as $0.58 \times 10^1 = 0.58E01$ where mantissa part is 0.58 (the part appearing before E) and exponent part is 1 (the part appearing after E). E01 represents 10^1 . The rule for writing a real literal in exponent form is given below :

A real literal in exponent form has two parts : a mantissa and an exponent. The mantissa must be either an integer or a proper real literal. The mantissa is followed by a letter E or e and the exponent. The exponent must be an integer.

The following are the valid real literals in exponent form :

152E05, 1.52E07, 0.152E08, 152.0E08, 152E+8, 1520E04, -0.172E-3

The following are invalid real literals in exponent form :

- (i) 172.E5 (At least a digit must follow the decimal point)
- (ii) 1.7E (No digit specified for exponent)
- (iii) 0.17E2.3 (Exponent can not have fractional part)
- (iv) 17,225E02 (No comma allowed)
- (v) .25E-7 (No preceding digits before decimal point)

(iii) Boolean Literals

The *boolean* type has two values, represented by the literals **true** and **false**, formed from ASCII letters.

A *boolean literal* is always of type boolean. It is either boolean value **true** or boolean value **false**.

(iv) Character Literals

A *character literal* is one character enclosed in single quotes, as in 'z'. The rule for writing character constant is given below :

A character literal in Java must contain one character and must be enclosed in single quotation marks.

Java allows you to have certain nongraphic characters in character constants. *Nongraphic characters* are those characters that cannot be typed directly from keyboard e.g., backspace, tabs, carriage return etc. These nongraphic characters can be represented by using escape sequences. An escape sequence is represented by a backslash (\) followed by one or more characters.

Following table (Table 5.1) gives a listing of escape sequences :

Table 5.1 Escape Sequences in Java

Escape Sequence	Nongraphic Character
\a	Audible bell (alert)
\b	Backspace
\f	Formfeed
\n	Newline or linefeed
\r	Carriage Return
\t	Horizontal tab
\v	Vertical tab
\\\	Backslash
\'	Single quote
\"	Double quote
\?	Question mark
\0n	Octal number (0n represents the number in octal)
\xHn	Hexadecimal number (Hn represents the number in hexadecimal)
\uHn	Unicode character represented through its hexadecimal code Hn.
\0	Null

In the above table, you see sequences representing \, ', " and ? also. Though these characters can be typed from the keyboard but when used without escape sequence, these carry a special meaning and have a special purpose, however, if these are to be typed as it is, then escape sequences should be used.

The following are examples of char literals :

'a', '%', '\t', '\\', '\\", '\u03a9', '\xFFFF', '\177', '\u00a9', '\u266a'

Unicode escapes are processed very early and thus must not be represented through \uHn for the sequences that have their unique representation escape sequences e.g., \n or \r etc. Because Unicode escapes are processed very early, it is not correct to write '\u000a' for a character literal whose value is linefeed (LF); the Unicode escape \u000a is transformed into an actual linefeed in translation and the linefeed becomes a *Line Terminator* and so the character literal is not valid. Instead, one should use the escape sequence '\n'. Similarly, it is not correct to write '\u000d' for a character literal whose value is carriage return (CR). Instead, use '\r'.

(v) String Literals

'Multiple Character' constants are treated as string-literals. The rule for writing string-literal is given below :

A string literal is a sequence of zero or more characters surrounded by double quotes. Each character may be represented by an escape sequence.

A string-literal is of class type *String*. Following are legal string literals :

"abc"	size 6 bytes (each character takes 2 bytes)
"\ab"	size 4 bytes (\a is one character)
"Seema\'s pen"	size 22 bytes
(\a and \' are escape sequences)	

(vi) The Null Literal

The null type has one value, the null reference, represented by the literal null, which is formed from ASCII characters. A *null literal* is always of the null type.

5.3.4 Separators

The following nine ASCII characters are the *separators* (punctuators) :

() { } [] ; ,

5.3.5 Operators

The following 37 tokens are the *operators*, formed from ASCII characters :

=	>	<	!	-	?	:	
=	<=	>=	!=	&&		++	--
/	-	*	/	&		^	%
<<	>>	>>>					
+=	-	*=	/=	&=	=	^=	%=
<<=	>>=	>>>=					

5.4 Concept of Data Types

Data can be of many types e.g. character, integer, real, string etc. Any thing enclosed in single quotes represents character data. Numbers without fractions represent integer data. Numbers with fractions represent real data and anything enclosed in double quotes represents a string. Since the data to be dealt with are of many types, a programming language must provide ways and facilities to handle all types of data.

Java like any other language provides ways and facilities to handle different types of data by providing *data types*.

Java data types are of *two* types :

- ◆ Primitive (or Intrinsic) data types
- ◆ Reference data types

Data Types are means to identify the type of data and associated operations of handling it.

Primitive datatypes come as a parts of the language. Java provides eight primitive datatypes, which are : *byte, short, int, long, float, double, char, boolean*.

Reference datatypes are constructed from primitive data types. These are : *classes, arrays and interface*. But their storage mechanism is different from primitive datatypes.

A reference datatype is used to store the memory address of an object. These datatypes generally store the memory addresses of a class or an array. The variables (data locations) storing memory addresses are known as *reference variables* and their *datatype* is known as a *reference datatype*. (See Fig. 5.1)

Let us now talk about primitive data type in details.

5.4.1 Primitive Datatypes

Primitives are the "basic" data values. The word 'Primitive' means *a fundamental component that may be used to create other larger parts*. Thus by primitive datatypes, we mean fundamental datatypes offered by Java.

Java supports following *four* categories of primitive datatypes :

- | | |
|--------------------------------------|---------------------------------|
| (i) Numeric Integral primitive types | (ii) Fractional primitive types |
| (iii) Character primitive types | (iv) Boolean primitive types. |

Let us learn more about these.

(i) Numeric Integral Types

The datatypes that are used to store numeric values fall under this sub category i.e., numeric primitive datatypes. There are *four* numeric integral types in Java.

◆ byte ◆ short ◆ int ◆ long

All *integral* numeric types store integer values i.e., whole numbers only. The storage sizes and the range of values supported by numeric integral types is being listed below in table 5.2.

Table 5.2 Integer Primitive Datatypes

Type	Size	Description	Range
byte	8 bits (1 byte)	Byte-length integer	-128 to +127
short	16 bits (2 bytes)	Short integer	-32,768 to +32,767
int	32 bits (4 bytes)	Integer	(about) -2 billion to + 2 billion i.e., -2^{31} to $2^{31} - 1$
long	64 bits (8 bytes)	Long integer	(about) -10E18 to +10E18 i.e., -2^{63} to $2^{63} - 1$

Please note that an l (small L) or L suffix on an integer means the integer is of *long* datatype e.g., 33L refers to a *long* integer value 33. Thus, we can say that 33 is an *int* value but 33L is a *long* value.

Depending upon your requirements, you can select appropriate datatype. Consider this – you need to store your phone no 27651000 in your program ? What do you think should be the datatype ? Can you store it in a *byte* or *short* datatype ? Well, you need to store it in *int* datatype. The reason being is that the number to be stored does not fall into ranges supported by *byte* and *short* types.

One more thing that you must know about datatypes that all datatypes are *signed* i.e., they can store negative as well as positive numbers.

Note Java does not support unsigned datatypes (i.e., only positive numbers). All numeric types in Java can store negative as well as positive numbers.

(ii) Fractional Numeric Types

The integral numbers covered above could store only whole numbers i.e., they could not have decimal values. Fractional datatypes can store fractional numbers i.e., numbers having decimal points. These are also called *floating-point datatypes*.

Following table displays various floating-point datatypes available in Java, along with their ranges.

Table 5.3 Floating Point Primitive Datatypes

Type	Size	Description	Range	Remarks
float	32 bits (4 bytes)	Single-precision floating point	-3.4 E + 38 to + 3.4 E + 38	Precision up to 6 digits. Examples : currency, temperature, percentage, length.
double	64 bits (8 bytes)	Double-precision floating point	-1.7 E + 308 to 1.7 E + 308	Precision up to 15 digits. Examples : large numbers or high precision, such as for astronomy or subatomic physics.

Now depending upon your requirements for the decimal precision of the data, you can select from **float** or **double** types.

For example, you can store the salary values of an employee using a data variable of the **float** type. To store decimal values with a higher degree of precision, such as values calculated using the functions, **sin()** and **sqrt()**, you use the **double** datatype.

By default, Java treats fractional numbers as of **double** datatype, e.g., if you write 0.234, it will be treated as a **double** value. To explicitly specify it to be of **float** type, use suffix **F** or **f**. i.e., write **1.234f** or **1.234F**. Similarly if you use suffix **d** or **D**, it means **double** i.e., **1.234D** or **1.234d** means it is a **double** value.

Note By default, Java assumes the fractional numbers to be of **double** datatype unless specified. To explicitly specify their type, use suffixes **F** and **D**.

Precision of Floating Point Numbers

Let us discuss some important things about precision of floating point numbers.

Consider writing the value $\frac{2}{3}$ in decimal notation :

0.66666666666666666666.....

The result keeps going, and going. There is no limit to the number of 6's required. With the **float** datatype, there are only 32 bits, not enough bits to represent an unlimited number of 6's.

Basically, the datatype **float** has 23 bits of precision. (The remaining bits of the 32 bits are used to indicate the size of the number.) This is equivalent to only about 7 decimal places.

The number of places of precision for **float** is the same no matter what the size of the number. Datatype **float** can represent numbers as big as about 3.4×10^{38} . But the precision of these large numbers will also be about 7 decimal digits e.g., 34.56789, 1.234567, 0.123456, 123.4567 etc., are all using the precision of 7 decimal digits.

Now try answering this question : What is wrong with the following constant, expected to be of type **float** ?

1230.00089F

What say? Nothing wrong with it ?? Well, read this : There are *nine* decimal places of precision. Datatype **float** can't handle that. (The compiler will round the number into a value that can fit in a **float**). You may argue that there are only five places used in the above number : the places used by the digits 1, 2, 3, 8, and 9. Unfortunately, the four 0's in the middle do count. It takes bits to represent them, even if they are zeros.

Primitive datatype **double** uses 64 bits, and has a much greater range, -1.7×10^{-308} to $+1.7 \times 10^{308}$. It also has a *much greater precision* : *about 15 significant decimal digits*.

Because of this, if you write a literal like 2.345 in a Java program, it will automatically be regarded as a **double**, even though a **float** might be good enough.

TIP

Use suffixes l, L, f, F, d or D to explicitly specify datatype of a data value.

(iii) Character Types

The character datatype – **char** datatype of Java – is used to store characters. A *character* in Java can represent all ASCII (*American Standard Code for Information Interchange*) as well as *Unicode* characters. The *Unicode* character representation code can represent nearly all languages of world such as *Hindi, English, German, French, Chinese, Japanese, Hebrew* etc. And since Java supports *Unicode*, it makes Java a truly universal programming language.

Because of *Unicode* characters¹, the size of **char** datatype of Java is 16 bits i.e., 2 bytes in contrast to 1 byte characters of other programming languages that support ASCII. Following table lists the size and range supported by a **char** datatype.

Table 5.4 Character Primitive Datatype

Type	Size	Description	Range	Remarks
char	16 bits (2 bytes)	Single character	0 to 65,536	Unicode characters (whereas other languages use 1-byte ASCII text characters) Examples : alphabets and numerals.

(iv) Boolean Type

Another of the primitive datatypes is the type **boolean**. It is used to represent a single true/false value. A **boolean** value can have only one of two values : *true* or *false*.

In a Java program, the words *true* and *false* always mean these **boolean** values. The data type **boolean** is named after a nineteenth century mathematician – *George Boole*, who discovered that a great many things can be done with true/false values. A special branch of algebra, *Boolean Algebra*, is based on Boolean values.

Following table 5.5 lists the size and range of **boolean** datatype.

Table 5.5 Boolean Primitive Datatype

Type	Size	Description	Range	Remarks
boolean	Java reserves 8 bits but only uses 1 bit.	Logical or boolean values	true or false [Note : These values are reserved words.]	Useful in logic test with if.

1. Unicode characters can be represented in 16 bits. ASCII characters can be represented in 8 bits.

Following table lists examples of various data items of different datatypes.

Here are some examples of literal values of various primitive types :

Table 5.6 Some data values and their types

Value	Datatype	Value	Datatype
178	int	26.77e3	double
8864L	long	'c'	char
37.266	double	true	boolean
37.266D	double	false	boolean
87.363F	float	22.34e2	double

5.4.2 Reference Datatypes

Broadly a reference in Java is a data element whose value is an address. Arrays, classes, and interfaces are *reference types*. The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable (Fig. 5.1).

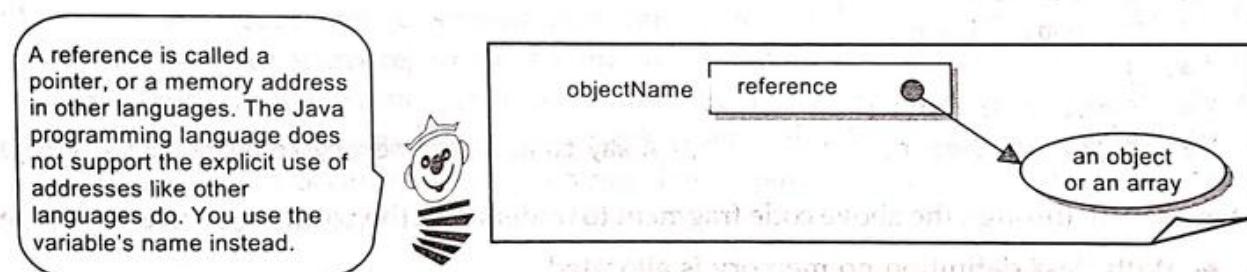


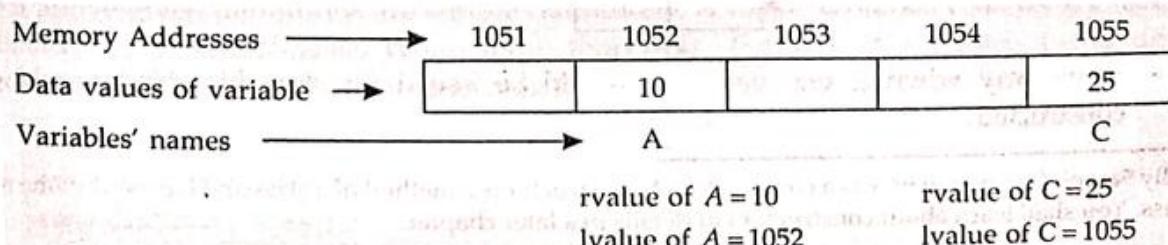
Figure 5.1 Reference datatype.

A reference is called a pointer, or a memory address in other languages. The Java programming language does not support the explicit use of addresses like other languages do. You use the variable's name instead.

Let us understand the difference between primitive data types and reference data types. Generally, there are *two* values associated with a symbolic variable :

1. Its data value, stored at some location in memory. This is sometimes referred to as a variable's *rvalue* (pronounced "are-value").
2. Its location value ; that is, the address in memory at which its data value is stored. This is sometimes referred to as a variable's *lvalue* (pronounced "el-value").

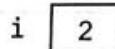
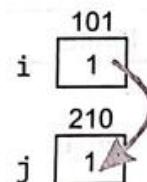
Following figure illustrates the concept of *rvalue* and *lvalue*. (The variable A has been stored at location number 1052).



Primitive data type operations deal only with **value i.e., actual read-value (rvalue)**. That is they store the read-value directly in them. For example, the following code shows that assigning primitive data variable *i* to another named *j* simply passes a copy of the value in *i* into *j*. That is, a data value passes and not a reference or pointer to a data location.

```
int i = 1;
int j = i; // Now j holds "1"
// that is, i's value (basically rvalue) passes to j
```

```
i = 2; // j still holds "1"
```



The reference types on the other hand, deal with the address *i.e.*, the location-value. They do not store the actual read-value, rather they store the address wherefrom actual data is obtained. Consider following code :

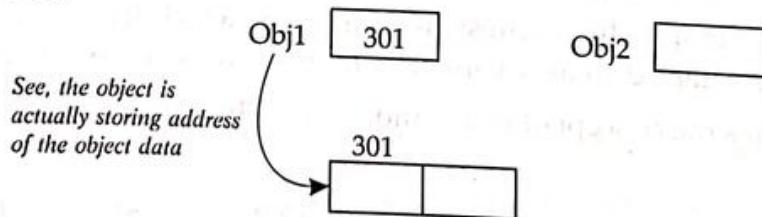
```
1. class Sample
2. {   public int a ;
3.     public int b ;
4. }
5. Sample obj1, obj2 ;
6. obj1 = new Sample( )2;           // A way to allocate memory to object namely obj1
```

Let us walk through the above code fragment to understand the working of reference types.

- ❖ With class definition no memory is allocated.
- ❖ With statement **Sample obj1, obj2 ;** enough memory is allocated to store a reference to an object (in this case objects *obj1* and *obj2* of *Sample* type)



- ❖ We cannot directly assign a numeric reference value to reference variable. If we want an object for the variable to refer to, we must use **new**. Thus, with statement **obj1 = new Sample();** enough memory is allocated for the actual object data *i.e.*, the object is constructed and its reference *i.e.*, memory address is put in the reference variable. See below :



- ❖ Same way when a **new** statement would be issued for *obj2*, this object would also be constructed.

-
2. Actually **Sample()** function here is a constructor. A constructor is a method of a class and has same name as that of the class. You shall learn about constructors in details in a later chapter.

Following Fig. 5.2 summarizes various datatypes in Java.

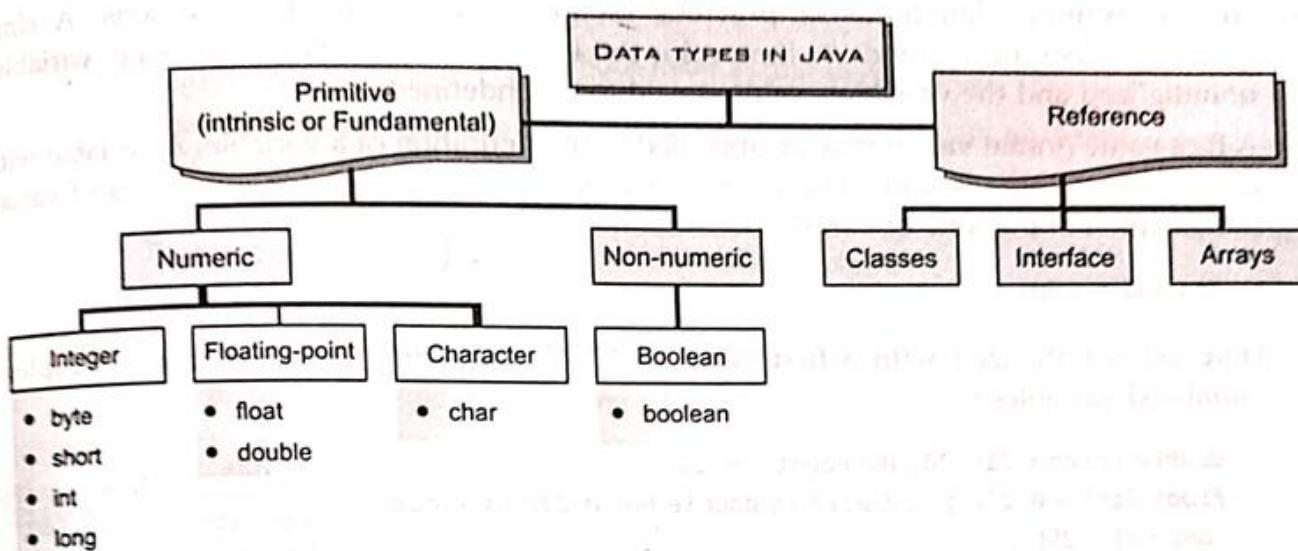


Figure 5.2 Datatypes in Java.

5.5 Variables

Variables represent named storage locations, whose values can be manipulated during program run. For instance, to store name of a student and marks of a student during a program run, we require storage locations that too named so that these can be distinguished easily. Variables, called as symbolic variables, serve the purpose. The variables are called symbolic variables because these are named. For instance, the following statement declares a variable *i* of the data type *int*:

```
int i ;
```

5.5.1 Declaration of a Variable

The declaration of a variable generally takes the following form

```
type variablename ;
```

where *type* is any Java data type and *variablename* is the name of the variable. A *variablename* is an identifier. Thus all rules of identifier naming apply to the name of a variable. Following declaration creates a variable *age* of *int* type:

```
int age ;
```

Following are some more examples of variable declarations

```
double pival ;
float res ;
```

A **Variable** is a named memory location, which holds a data value of a particular data type.

The above declaration creates a variable *pival* of type *double* and a variable *res* of type *float*.

All above given definitions are simple definitions. A *simple definition* consists of a *type specifier* followed by a *variable-name*. When more than one identifier of a type is being defined, a comma-separated list of identifiers may follow the type specifier. For example,

```
double salary, wage ;
int month, day, year ;
Long distance, area ;
```

5.5.2 Initialization of Variables

All the example definitions of previous section 5.5.1 are simple definitions. A simple definition does not provide a first value or initial value to the variable *i.e.* variable is uninitialized and the variable's value is said to be undefined.

A first value (initial value) may be specified in the definition of a variable. A variable with a declared first value is said to be an *initialised* variable. Java supports two forms of variable initialization at the time of variable definition :

```
int val = 1001 ;
```

Here *val* is initialized with a first value of 1001³. Following are some more examples of initialized variables :

```
double price = 214.70, discount = 0.12 ;
float fint = 0.27F ; // 0.27 is double but 0.27F is float
long val = 25L ;
```

Now consider the following example program that declares variables of different datatypes and prints their values.

Note While naming variables, make sure to follow identifier naming rules and conventions.

Example 5.1. Write a Java program that initializes three variables namely *hoursWorked*, *payRate* and *taxRate* and then calculates and prints payment amount and Tax payable as *hoursWorked* × *payRate* and Payment amount × *taxRate* respectively.

Solution.

```
class Example {
    public static void main (String[ ] args) {
        long hoursWorked = 50 ;
        double payRate = 40.0, taxRate = 0.10 ;
        System.out.println("Hours Worked :" + hoursWorked) ;
        System.out.println("Payment Amount :" + (hoursWorked * payRate)) ;
        System.out.println("Tax Payable :" + (hoursWorked * payRate * taxRate)) ;
    }
}
```

The output of above example will be as :

```
Hours worked :50
Payment Amount :2000.0
Tax Payable :200.0
```

The character * means *multiply*. In the program, (*hoursWorked* * *payRate*) means to multiply the number stored in *hoursWorked* by the number stored in *payRate*.

When it follows a character string, + means to add characters to the end of the character string. So "Hours Worked :" + hours Worked makes a character string starting with "Hours Worked :" and ending with characters for the value of hours Worked.

- This type of initialization can take place either inside a method or inside a class provided the declaration also includes keyword static e.g.,

```
static int val = 1001 ;
```

Notes about main() Program

Two more things that you can know about main at this stage are :

- (i) main() can be declared as public static void... (as we did in example 5.1) or as static public void...
- (ii) the variable name inside main's () can be other than args as long as the type is String []. That is, even following statements would also be right.
 - ¤ static public void main (String list []) ;
 - ¤ static public void main (String [] strlist) ;
 - ¤ static public void main (String [] words) ;

Dynamic Initialization

The expression that initializes a variable (that assigns it a value for initial use) can be an expression with :

- ♦ a literal e.g.,

```
byte a = 3 ;
```

- ♦ a reference to another variable e.g.,

```
short a = 0 ;
short b = a ;           //initialized with value of another variable.
```

- ♦ a call to a method, in which case the return value determines the initialization. This type of initialization is called *dynamic initialization*.

Following example program 5.2 illustrates the usage of dynamic initialization.

Example 5.2. A Java program to illustrate dynamic initialization of variables.

Solution.

```
public class Ex9_2 {
    public static void main(String[ ] args) {
        // initialize a and b, but not yet c
        double a = 3.0, b=4.0 ;
        // c is dynamically initialized by the return value
        // of the square root method (sqrt) of the build-in Math class
        double c= Math.sqrt(a*a + b*b) ;
        System.out.println("Hypotenuse is " +c) ;
    }
}
```

The output of above example will be as :

Hypotenuse is 5.0

Note Do you know that variables can have same name as a method / function or class ? However, you should always avoid doing so.

Initial Values of Variables

Every variable in a program must have a value before its value is used :

- ◆ Each class variable (also called Field variable), instance variable, or array component is initialized with a *default value* when it is created :

Type	Initial / Default value
byte	0 (Zero) of byte type
short	0 (Zero) of short type
int	0
long	0L
float	0.0F
double	0.0D
char	null character i.e., '\u0000'
boolean	false
All reference types	null

Variable Scope

One important thing that you must know about variable is their *scope*. *Scope* generally refers to the program-region within which a variable is accessible. The broad rule is a variable is accessible within the set of braces it is declared in e.g.,

```
{
    int a;
    :      /* a would be accessible as long as its block (a block is marked
           with a pair of matching brace) is not closed. Variable a is said to
           have block scope */
}
```

The scope rules shall be covered in details in chapter 12. But right now, you can have a look at following example program 5.3 that demonstrates **block scope**.

Example 5.3. Program to demonstrate block scope of variables.

Solution.

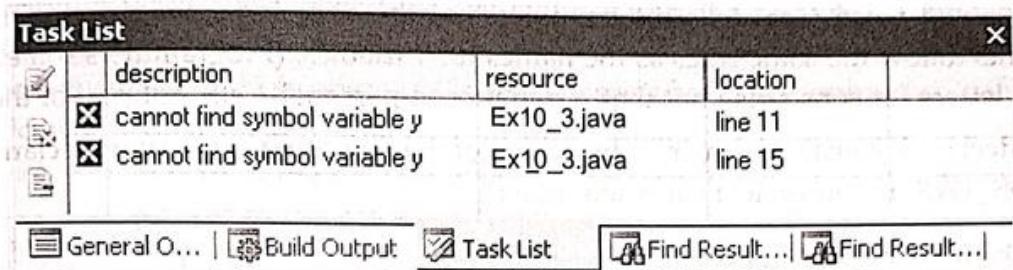
```
public class Ex10_3 {
    public static void main( String [ ] args )
    {
        int x ;
        x=10 ;
        if (x==10)
        {
            int y = 20 ;
            //start new scope
            //known to all code within this function
            //known only to this block
            //x and y both known here.
            x = y * 2 ;
        }
        y=100 ;
        //Error!! y not known here
    }
}
```

```

//x is still known here.
System.out.println("x is " + x);           // No problem here
//the compiler complains if you reference a variable out of scope
System.out.println("x and y : " + x + " " +y); // will produce error
}
}

```

Compilation Result



There can be a common error involving the concept of scope :

```

if (.....) {
    int myInteger = 17 ;
    .....
}

System.out.println("The value of myInteger =" + myInteger) ; //error

```

The final line won't compile because the local variable *myInteger* is out of scope. The scope of *myInteger* is the block of code between the { and }.

The *myInteger* variable does not exist after the closing curly brace {} ends that block of code.

Now that you know about scope of variables, you should also know that only class variables (field variables) are automatically initialized with their type's default values ; local variables must be explicitly initialized.

Note Only class (field) variables automatically get initialized with their type's default values ; local variables are not initialized automatically.

5.6 Constants

Often in a program you want to give a name to a constant value. For example, you might have a fixed *tax rate* of 0.030 for *goods* and a *tax rate* of 0.020 for *services*. These are constants, because their value is not going to change when the program is executed. It is convenient to give these constants a name.

This can be done as follows :

```
final double TAXRATE = 0.25 ;
```

The keyword **final** makes a variable as constant i.e., whose value can not be changed during Program execution.

Consider the following program that declares and uses two constants.

```
class CalculateTax {
    public static void main (String[ ] arg) {
        final double GOODS_TAX = 0.030 ;
        final double SERVICE_TAX = 0.020 ;
        ....
    }
}
```

The reserved word **final** tells the compiler that the value will not change. The names of constants follow the same rules as the names for variables. (Programmers sometimes use all capital letters for constants ; but that is a matter of personal style, not part of the language).

Once declared constants, their value can not be modified e.g., after declaring constant **GOODS_TAX**, if you issue a statement like :

```
GOODS_TAX = 0.050 ; //error
```

it will cause an error, as the value of constants cannot be modified.

Note The reserved word **final** tells the compiler that the value will not change.

Advantages of Constants

There are *two* big advantages to using constants :

1. They make your program easier to read and check for correctness.
2. If a constant needs to be changed (for instance a new tax law changes the rates) all you need to do is change the declaration. You don't have to search through your program for every occurrence of a specific number.



Let Us Revise

- ❖ Character set is a set of valid characters that a language can recognise. Java uses Unicode character set.
- ❖ The smallest individual unit in a program is known as a token.
- ❖ Java offers five tokens : keywords, identifiers, literals, punctuators and operators.
- ❖ Keyword is a word carrying special meaning and purpose.
- ❖ Identifiers are user defined names for different parts of program. In Java, identifiers can have alphabets, digits, underscore and dollar sign characters ; must not be keyword or boolean literal or null literal ; must not begin with a digit ; and can be of any length.
- ❖ Literals are data items that never change their value during a program run.
- ❖ Java allows following literals :
 - integer-literal (Decimal, Octal, Hexadecimal) ; character-literal ; floating-literal ; string-literal
- ❖ Data types are means to identify the type of data and associated operations of handling it.
- ❖ Java provides two types of data types : primitive and reference data types.
- ❖ Java supports eight primitive data types : byte, short, int, long, float, double, char and boolean.
- ❖ A variable is a named memory location, which holds a data value of a particular data types.
- ❖ The region of program within which a variable is accessible is called its scope.
- ❖ Memory locations whose values can not be changed within the program are called constants.
- ❖ Keyword **final** while declaring a variable, makes it a constant.

5.7 Operators in Java

The operations being carried out are represented by operators.

Java's rich set of operators comprises of arithmetic, relational, logical and certain other type of operators. Let us discuss these operators in detail.

The operations (specific tasks) are represented by operators and the objects of the operation(s) are referred to as operands.

5.7.1 Arithmetic Operators

To do arithmetic, Java uses operators. It provides operators for *five* basic arithmetic calculations : addition, subtraction, multiplication, division and remainder which are $+$, $-$, $*$, $/$ and $\%$ respectively. Each of these operators is a *binary operator* i.e., it requires two values (operands) to calculate a final answer. Apart from these binary operators, Java provides two *unary arithmetic operators* (that require one operand) also which are unary $+$, and unary $-$.

5.7.1A Unary Operators

1. Unary +. The operator unary ' $+$ ' precedes an operand. The operand (the value on which the operator operates) of the unary $+$ operator must have *arithmetic type* and the result is the value of the argument. For example,

```
if a = 5 then +a means 5.  
if a = 0 then +a means 0.  
if a = -4 then +a means - 4.
```

Operators that act on one operand are referred to as **Unary Operators**.

2. Unary -. The operator unary $-$ precedes an operand. The operand of the unary $-$ operator must have *arithmetic type* and the result is the negation of its operand's value. For example,

```
if a = 5 then -a means - 5.  
if a = 0 then -a means 0 (there is no quantity known as - 0)  
if a = -4 then -a means 4.
```

This operator reverses the sign of the operand's value.

5.7.1B Binary Operators

The operands of a binary operator are distinguished as the *left* or *right* operand. Together, the operator and its operands constitute an expression.

Operators that act upon two operands are referred to as **Binary Operators**.

1. Addition operator $+$. The arithmetic binary operator $+$ adds values of its operands and the result is the sum of the values of its two operands. For example,

```
4 + 20 results in 24.  
a + 5 (where a = 2) results in 7.  
a + b (where a = 4, b = 6) results in 10.
```

Its operands may be of integer (byte, short, char, int, long) or float (float, double) types.

2. Subtraction operator $-$. The $-$ operator subtracts the second operand from the first. For example,

```
14 - 3 evaluates to 11.  
a - b where a = 7, b = 5) evaluates to 2.
```

The operands may be of integer or float types.

3. Multiplication operator *. The * operator multiplies the values of its operands. For example,

$3 * 4$ evaluates to 12.

$b * 4$ (where $b = 6$) evaluates to 24.

$a * c$ (where $a = 3, c = 5$) evaluates to 15.

The operands may be of integer or float types.

4. Division operator /. The / operator divides its first operand by the second. For example,

$100/5$ evaluates to 20.

$a/2$ ($a = 16$) evaluates to 8.

a/b ($a = 15.9, b = 3$) evaluates to 5.3.

The operands may be of integer or float types.

5. Modulus operator %. The % operator finds the modulus of its first operand relative to the second. That is, it produces the *remainder* of dividing the first by the second operand. e.g.,

$19 \% 6$ evaluates to 1, since 6 goes into 19 three times with a remainder 1.

Similarly, $7.6 \% 2.9$ results into 1.8 and $-5 \% -2$ results into -1.

Operator + With Strings

You have used the operator '+' with numbers. When you use + with numbers, the result is also a number. However, if you use operator + with strings, it concatenates them e.g.,

$5 + 6$ results into 11.

"5" + "6" results into "56".

"17" + "A, V. Vihar" results into "17 A, V. Vihar"

"abc" + " 123" results into "abc 123"

$5 + "xyz"$ results into "5xyz"

(Java would internally convert 5 into "5" first and then concatenate it with "xyz".)

5.7.2 Increment/Decrement Operators (++ , --)

Java includes two useful operators not generally found in other computer languages (except C and C++). These are the increment and decrement operators, ++ and --. The operator ++ adds 1 to its operand, and -- subtracts one. In other words,

$a = a + 1 ;$

is the same as

$++a ;$ or $a++ ;$

and

$a = a - 1 ;$

is the same as

$--a ;$ or $a-- ;$

However, both the increment and decrement operators come in two varieties : they may either precede or follow the operand. The *prefix* version comes before the operand (as in $++a$ or $--a$) and the *postfix* version comes after the operand (as in $a++$ or $a--$). The two versions have the same effect upon the operand, but they differ when they take place in an expression.

Working with prefix version

When an increment or decrement operator precedes its operand (*i.e.*, in its prefix form), Java performs the increment or decrement operation *before* using the value of the operand. For example, the expression

`sum = sum + (++count) ;`

will take place in the following fashion. (Assuming the initial values of *sum* and *count* are 0 and 10 respectively).

Prefix Version's Working	<i>Sum</i>		<i>Count</i>		(Principal : Change-then-use)	
	1. -		0			
	2. -		0			
	3. 11	=	0	+ 11		
					← Now use it	

The expression

`P = P * --N ;`

will take place in the following fashion. (Assuming the initial values of *P* and *N* are 4 and 8 respectively).

Prefix Version's Working	<i>P</i>		<i>N</i>		(Principal : Change-then-use)	
	1. -		8			
	2. -		7			
	3. 28	=	4	* 7		
					← Now use it	

Working with postfix version

When an increment or decrement operator follows its operand (*i.e.*, in its postfix form), Java first uses the value of the operand in evaluating the expression before incrementing or decrementing the operand's value. For example, the expression

`sum = sum + count++ ;`

will take place in the following fashion. (Assuming the initial values of *sum* and *count* are 0 and 10 respectively).

Prefix Version's Working	<i>Sum</i>		<i>Count</i>		(Principal : Change-then-use)	
	1. -		10			
	2. 10		10			
	3. 10	=	0	+ 11		
					← Now increment it	

The expression

`P = P * N-- ;`

Note The prefix increment or decrement operators follow change-then-use rule *i.e.*, they first change (increment or decrement) the value of their operand, then use the new value in evaluating the expression.

will take place in the following fashion. (Assuming the initial values of P and N are 4 and 8 respectively).

	P	P	N	(Principal : Change-then-use)
Prefix Version's Working	1. $\begin{array}{ c } \hline - \\ \hline \end{array}$	$=$	$\begin{array}{ c } \hline 4 \\ \hline 4 \\ \hline 4 \\ \hline \end{array}$	Initial values
2. $\begin{array}{ c } \hline 32 \\ \hline \end{array}$	$=$	$\begin{array}{ c } \hline 4 \\ \hline 4 \\ \hline 4 \\ \hline \end{array}$	\leftarrow First use it	
3. $\begin{array}{ c } \hline 32 \\ \hline \end{array}$	$=$	$\begin{array}{ c } \hline 8 \\ \hline 8 \\ \hline 7 \\ \hline \end{array}$	\leftarrow Now decrement it	

Please notice that the overall effect on the operand's value, in both the cases of prefix or postfix operators, is the same (the final value of $++$ count & count $++$ and $--N$ & $N--$ are the same). However, they differ in when they take place in evaluation of an expression.

Example 5.4. Evaluate $x = ++y + 2y$ if $y = 6$.

Solution.

Initially $y = 6$

$$\begin{aligned} &= 7 + 2 \times 7 \quad (++y = 7 \text{ and after this} \\ &\qquad\qquad\qquad y \text{ also becomes } 7) \\ &= 7 + 14 = 21 \end{aligned}$$

Thus $x = 21$.

Note The postfix increment or decrement operators follow **use-then-change** rule i.e., they first use the value of their operand in evaluating the expression and then change (increment or decrement) the operand's value.

Note The increment operator $++$ and decrement operator $--$ are unary operators i.e., they operate upon single operand. And the postfix increment/decrement operators have higher precedence over prefix increment/decrement operators.

5.7.3 Relational Operators

In the term *relational operator*, relational refers to the relationships that values (or operands) can have with one another. Thus, the relational operators determine the relation among different operands. Java provides six relational operators for comparing numbers and characters. But they don't work with strings. If the comparison is true, the relational expression results into the boolean value *true* and to boolean value *false*, if the comparison is false. The six relational operators are :

$<$ (less than),	$<=$ (less than or equal to),	$= =$ (equal to)
$>$ (greater than),	$>=$ (greater than or equal to)	$!=$ (not equal to)

Table 5.7 summarizes the action of these relational operators.

Table 5.7 Relational Operators

p	q	$p < q$	$p \leq q$	$p == q$	$p > q$	$p \geq q$	$p != q$
0	1	t	t	f	f	f	t
1	0	f	f	f	t	t	f
3	3	f	t	t	t	t	f
2	6	t	t	f	f	f	t

t represents *true* and f represents *false*.

The relational operators have a lower precedence than the arithmetic operators. That means the expression

$a + 5 > c - 2$... expression 1

corresponds to

$(a + 5) > (c - 2)$... expression 2

and not the following

$a + (5 > c) - 2$ expression 3

Expression 1 means the expression 2 and not the expression 3.

Though relational operators are easy to work with, yet while working with them, sometimes you get unexpected results and behaviour from your program. To avoid so, I would like you to know certain tips regarding relational operators.

TIP

Do not confuse the = and the == operators.

A very common mistake is to use the assignment operator = in place of the relational operator ==. Do not confuse the testing the operator == with the assignment operator (=). For instance, the expression

`value == 3`

tests whether *value* is equal to 3? The expression has the boolean value *true* if the comparison is true and boolean false if it is false.

But the expression

`value = 3`

assigns 3 to *value*. The whole expression, in this case, has the value 3 because that's the value of the left-hand side.

Another tip is regarding floating-point numbers.

TIP

Avoid equality comparisons on floating-point numbers.

Floating-point arithmetic is not as exact and accurate as the integer arithmetic is. For instance, $3 * 5$ is exactly 15, but $3.25 * 5.25$ is nearly equal to 17.06 (if we are working with number with 2 decimal places). The exact number resulting from $3.25 * 5.25$ is 17.0625. Therefore, after any calculation involving floating-point numbers, there may be a small residue error. Because of this error, you should avoid the equality and inequality comparisons on floating-point number.

The relational operators group left-to-right i.e., $a < b < c$ means $(a < b) < c$ and not $a < (b < c)$.

5.7.4 Logical Operators

Relational operators often are used with logical operators (also known as *conditional operators* sometimes) to construct more complex decision-making expressions. The Java programming language supports six conditional operators—five binary and one unary – as shown in the following Table 5.8.

Table 5.8 Logical operators

Operator	Use	Returns true if
<code>&&</code>	<code>op1 && op2</code>	<code>op1</code> and <code>op2</code> are both <i>true</i> , conditionally evaluates <code>op2</code>
<code> </code>	<code>op1 op2</code>	either <code>op1</code> or <code>op2</code> is <i>true</i> , conditionally evaluates <code>op2</code>
<code>!</code>	<code>! op</code>	<code>op</code> is <i>false</i> . Can you make out this operator is unary operator?
<code>&</code>	<code>op1 & op2</code>	<code>op1</code> and <code>op2</code> are both <i>true</i> , always evaluates <code>op1</code> and <code>op2</code>
<code> </code>	<code>op1 op2</code>	either <code>op1</code> or <code>op2</code> is <i>true</i> , always evaluates <code>op1</code> and <code>op2</code>
<code>^</code>	<code>op1 ^ op2</code>	if <code>op1</code> and <code>op2</code> are different – that is if one or the other of the operands is <i>true</i> but not both

Let us examine them now.

The logical OR operator `||`

The logical OR operator (`||`) combines two expressions which make its operands. The logical OR ("`||`") operator evaluates to *true* if either of its operands evaluate to *true*.

This principle is used while testing evaluating expressions. Following are some examples of logical OR operation :

<code>(4 == 4) (5 == 8)</code>	results into <i>true</i> because first expression is <i>true</i> .
<code>1 == 0 0 > 1</code>	results into <i>false</i> because neither expression is <i>true</i> (both are <i>false</i>).
<code>5 > 8 5 < 2</code>	results into <i>false</i> because both expressions are <i>false</i> .
<code>1 < 0 8 > 0</code>	results into <i>true</i> because second expression is <i>true</i> .

The operator `||` (logical OR) has lower precedence than the relational operators, thus, we don't need to use parenthesis in these expressions.

The logical AND operator (`&&`)

The logical AND operator, written as `&&`, also combines two expressions into one. The resulting expression has the value *true* only if both of the original expressions (its operands) are *true*. Following are some examples of AND operator (`&&`) :

<code>(6 == 3) && (4 == 4)</code>	results into <i>false</i> because first expression is <i>false</i> .
<code>(4 == 4) && (8 == 8)</code>	results into <i>true</i> because both expressions are <i>true</i> .
<code>6 < 9 && 4 > 2</code>	results into <i>true</i> because both expressions are <i>true</i> .
<code>6 > 9 && 5 < 2</code>	results into <i>false</i> because both expressions are <i>false</i> .

Because logical AND operator (`&&`) has lower precedence than the relational operators, we don't need to use parentheses in these expressions.

The logical NOT operator (`!`)

The logical NOT operator, written as `!`, works on single expression or operand i.e., it is a *unary operator*. The logical NOT operator (`!`) negates or reverses the truth value of the expression following it i.e., if the expression is *true*, then `!expression` is *false*, and vice versa.

Following are some examples of logical NOT operation :

<code>! (5 != 0)</code>	results into false because 5 is non zero (i.e., true)
<code>! (5 > 2)</code>	results into false because the expression $5 > 2$ is true.
<code>! (5 > 9)</code>	results into true because the expression $5 > 9$ is false.

The logical negation operator ! has a higher precedence than any of the relational or arithmetic operators. Therefore, to negate an expression, you should enclose the expression in parentheses :

`! (x > 5)` will reverse the result of the expression $x > 5$ whereas `! x > 5` is equivalent to `(! x) > 5`

i.e., it will first reverse the truth value of x and then test whether the reverse of x 's truth value is greater than 5 or not.

Operators &, | and ^

When both operands are *boolean*, the operator & performs the same operation as &&. However, & always evaluates both of its operands and returns *true* if both are *true*.

Likewise, when the operands are *boolean*, | performs the same operation as ||. The | operator always evaluates both of its operands and returns *true* if at least one of its operands is *true*, which is contrary to the functioning of || that evaluates first operand first and if it evaluates to *true*, gives the result as *true* without evaluating the second operand. When their operands are numbers, & and | perform bitwise manipulations.

Operator ^ returns *true* if its operands are different from one another i.e., if one operand is *true*, the other should be *false* and vice-versa.

Note Operators & and | perform bitwise operations if their operands are *numbers*, otherwise for *boolean values* they perform similar to conditional operators && and ||.

5.7.5 Shift Operators

A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left. This table summarizes the shift operators available in the Java programming language.

Table 5.9 Shift Operators

Operator	Use	Operation
>>	<code>op1 >> op2</code>	shift bits of <i>op1</i> right by distance <i>op2</i> (signed shifting)
<<	<code>op1 << op2</code>	shift bits of <i>op1</i> left by distance <i>op2</i> (signed shifting)
>>>	<code>op1 >>> op2</code>	shift bits of <i>op1</i> right by distance <i>op2</i> (unsigned shifting)

Each operator shifts the bits of the left-hand operand over by the number of positions indicated by the right-hand operand. The shift occurs in the direction indicated by the operator itself. For example, the following statement shifts the bits of the integer 13 to the right by one position :

`13 >> 1 ;`

The binary representation of the number 13 is 1101. The result of the shift operation is 1101 shifted to the right by one position i.e., 110, or 6 in decimal. The left-hand bits are filled with 0s as needed.

Populating Vacated Bits for Shift Operations

In Java, the *signed* right shift operation populates the vacated bits with the *sign bit*, while the left shift and the *unsigned* right shift populate the vacated bits with zeros.

To understand this, consider the examples shown in Fig. 5.3.

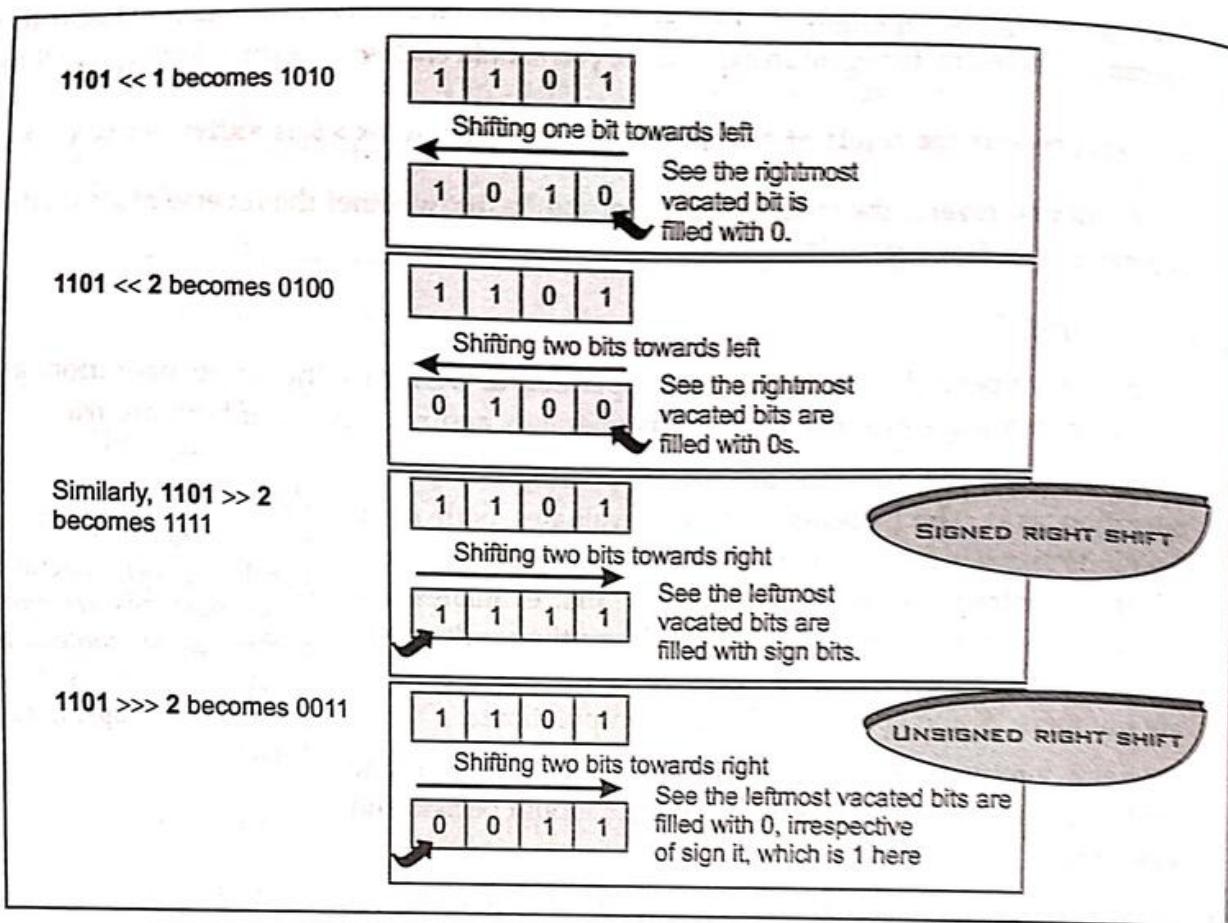


Figure 5.3 Populating vacated bits for shift operations.

In the case of the *left shift* \ll and *unsigned right-shift* $\gg>$ operators, the new bits are set to zero. However, in the case of the *signed right-shift* \gg operator, the new bits take the value of the *most significant bit (msb)* before the shift. When the *msb* is 1 (indicating negative number) prior to shift, the new bits added in the left are 1's. Conversely, when 0 bit is *msb* prior to shift, 0 bits are introduced during the shift.

In case of unsigned shift, if left hand operator is negative, then the result is equivalent to *left operand right shifted by right operand plus 2 left-shifted by inverted value of right operand*.

For example, $-16 \gg> 2 = (-16 \gg 2) + (2 \ll \sim 2) = 1,073,741,820$.

Another important thing that you need to know about shift operators is that when the value to be shifted (the left operand) is an int, only last 5 digits of the right hand operand are used to perform the shift. In other words, the actual size of the shift is the value of right-hand operand masked by 31 (i.e., right-hand-operand & 31). That is, the shift distance is always between 0 and 31 for integers. (if shift-value is ≥ 32 , shift is 32% value)

For example, if we have to calculate $16 \gg 34$, then first of all the right operand (which is 34 here) will be checked whether it is ≥ 32 . 34 is > 32 . Now it will be masked with 31 i.e., 34 & 31 and the result will give us the shift value which will be then applied on left operand.

34 =	00000000 00000000 00000000 00100010
31 =	00000000 00000000 00000000 00011111

and

shift value = 00000000 00000000 00000000 00000010 ($= 2_{10}$)

$$\begin{aligned} \therefore 16 \gg 34 &= 16 \gg 2 = \\ &\quad 00000000 \quad 00000000 \quad 00000000 \quad 00010000 \\ &\quad \gg 2 \\ &= 00000000 \quad 00000000 \quad 00000000 \quad 00000100 \quad = 4_{10} \\ \therefore 16 \gg 34 &= 16 \gg 2 = 4 \end{aligned}$$

Similarly, if we have to compute $16 \ll -29$ then

$-29 =$	11111111 11111111 11111111 11100011	(2's complement of 29)
31 =	00000000 00000000 00000000 00011111	

and

$$\begin{aligned} \text{shift value} &= 00000000 \quad 00000000 \quad 00000000 \quad 00000011 \\ \therefore 16 \ll -29 &= 16 \ll 3 = \\ &\quad 00000000 \quad 00000000 \quad 00000000 \quad 00010000 \\ &\quad \ll 3 \\ &= 00000000 \quad 00000000 \quad 00000000 \quad 10000000 \quad = 128_{10} \end{aligned}$$

$\therefore 16 \ll -29 = 128$.

If the left operand (the value to be shifted) is of type long, then only the last 6 digits of the right hand operand are used to perform the shift. That is, the actual shift size is within 0-63 and if shift value ≥ 64 then shift value is 64% value (i.e., right-hand operator masked by 63).

Note In case of an int, only the last 5 digits of the right operand determine the actual shift value and in case of a long, the last 6 digits of the right operand determine the actual shift value.

5.7.6 Bitwise Operators

The bitwise operators provided by Java are : & (bitwise AND), ^ (eXclusive_OR i.e., XOR), | (OR) operations and ~ (bitwise complement). The bitwise operators work with integral types i.e., byte, short, int and long types. The bitwise operators are sometimes referred to as *logical operators* when they work with boolean values.

Following Table 5.10 shows the four operators the Java programming language provides to perform bitwise functions on their operands :

Table 5.10 Bitwise operators

Operator	Use	Operation
&	$op1 \& op2$	bitwise and
	$op1 op2$	bitwise or
^	$op1 ^ op2$	bitwise xor
~	$\sim op2$	bitwise complement

The bitwise operations calculate each bit of their results by comparing the corresponding bits of the two operands on the basis of these *three* rules :

- ◆ For AND operations, 1 AND 1 produces 1. Any other combination produces 0.
- ◆ For XOR operations, 1 XOR 0 produces 1, as does 0 XOR 1. (All these operations are commutative.) Any other combination produces 0.
- ◆ For OR operations, 0 OR 0 produces 0. Any other combination produces 1.

Let us examine them one by one.

5.7.6A The AND operator &

When its operands are numbers, the & operation performs the bitwise AND function on each parallel pair of bits in each operand. The AND function sets the resulting bit to 1 if the corresponding bit in both operands is 1, as shown in the following table 5.11.

Table 5.11 The bitwise AND (&) operation

op1	op2	Result
0	0	0
0	1	0
1	0	0
1	1	1

Suppose that you were to AND the values 13 and 12, like this : 13 & 12. The result of this operation is 12 because the binary representation of 12 is 1100, and the binary representation of 13 is 1101.

$$\begin{array}{r}
 1101 \\
 & //13 \\
 \& 1100 \\
 & //12 \\
 \hline
 1100
 \end{array}$$

If both operand bits are 1, the AND function sets the resulting bit to 1 ; otherwise, the resulting bit is 0. So, when you line up the two operands and perform the AND function, you can see that the two high-order bits (the two bits farthest to the left of each number) of each operand are 1. Thus, the resulting bit in the result is also 1. The low-order bits evaluate to 0 because either one or both bits in the operands are 0.

5.7.6B The inclusive OR operator |

When both of its operands are numbers, the | operator performs the inclusive OR operation. Inclusive OR means that if either of the two bits is 1, the result is 1. The following table 5.12 shows the results of inclusive OR operations :

Table 5.12 *The inclusive OR (|) operation*

<i>op1</i>	<i>op2</i>	<i>Result</i>
0	0	0
0	1	1
1	0	1
1	1	1

5.7.6C The eXclusive OR (XOR) operator ^

Exclusive OR means that if the two operand bits are different, the result is 1 ; otherwise the result is 0. The following table 5.13 shows the results of an eXclusive OR operation.

Table 5.13 *The eXclusive OR (^) operation*

<i>op1</i>	<i>op2</i>	<i>Result</i>
0	0	0
0	1	1
1	0	1
1	1	0

5.7.6D The complement operator ~

The complement operator inverts the value of each bit of the operand : if the operand bit is 1 the result is 0 and if the operand bit is 0 the result is 1.

Table 5.14 *The complement (~) operation*

<i>op1</i>	<i>Result</i>
0	1
1	0

5.7.7 Assignment Operators

Like other programming languages, Java offers an assignment operator =, to assign one value to another e.g.,

```
int x, y, z ;
x = 9 ;
y = 7 ;
z = x + y ;
z = z * 2;
```

Java Shorthand Operators

Java offers special shorthands that simplify the coding of a certain type of assignment statement. For example,

`a = a + 10 ;`

can be written as

`a += 10 ;`

The operator pair `+=` tells the compiler to assign to `a` the value of `a + 10`. This shorthand works for all the binary operators in Java (those that require two operands). The general form of Java shorthand is

`var = var operator expression`

is same as

`-var operator = expression`

Following are some examples of Java shorthands :

<code>x -= 10 ;</code>	equivalent to	<code>x = x - 10 ;</code>
<code>x *= 3 ;</code>	equivalent to	<code>x = x * 3 ;</code>
<code>x /= 2 ;</code>	equivalent to	<code>x = x / 2 ;</code>
<code>x %= z ;</code>	equivalent to	<code>x = x % z ;</code>

Thus, we can say `=, *, /, %, +=, -=` are assignment operators in Java. The operators `*, /, %, +=, -=` are called *arithmetic assignment operators*. One *important* and *useful* thing about such arithmetic assignment operators of Java is that they *combine* an arithmetic operator and an assignment operator, and eliminate the repeated operand thereby facilitate a *condensed approach*.

The following table 5.15 lists the shortcut assignment operators and their lengthy equivalents :

Table 5.15 Shorthand Assignment operators

Operator	Use	Equivalent to
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>
<code>&=</code>	<code>op1 &= op2</code>	<code>op1 = op1 & op2</code>
<code> =</code>	<code>op1 = op2</code>	<code>op1 = op1 op2</code>
<code>^=</code>	<code>op1 ^= op2</code>	<code>op1 = op1 ^ op2</code>
<code><<=</code>	<code>op1 <<= op2</code>	<code>op1 = op1 << op2</code>
<code>>>=</code>	<code>op1 >>= op2</code>	<code>op1 = op1 >> op2</code>
<code>>>>=</code>	<code>op1 >>>= op2</code>	<code>op1 = op1 >>> op2</code>

5.7.8 Other Operators

The following table 5.16 lists the other operators that the Java programming language supports.

Table 5.16 Other Operators

Operator	Description
? :	Shortcut if-else statement
[]	Used to declare arrays, create arrays, and access array elements
.	Used to form qualified names
(params)	Delimits a comma-separated list of parameters
(type)	Casts (converts) a value to the specified type
new	Creates a new object or a new array
instanceof	Determines whether its first operand is an instance of its second operand

5.7.8A Conditional operator ? :

Java offers a shortcut conditional operator (?:) that stores a value depending upon a condition. This operator is **ternary operator** i.e., it requires *three operands*. The general form of conditional operator ?: is as follows :

expression1 ? *expression2* : *expression3*

If *expression1* evaluates to true i.e., 1, then the value of the whole expression is the value of *expression2*, otherwise, the value of the whole expression is the value of *expression3*. For instance,

```
result = marks >= 50 ? 'P' : 'F' ;
```

The identifier *result* will have value 'P' if the test expression ≥ 50 evaluates to true (1) otherwise *result* will have value 'F'. Following are some more examples of conditional operator ?:

$6 > 4 ? 9 : 7$ evaluates to 9 because test expression $6 > 4$ is true.

$4 == 9 ? 10 : 25$ evaluates to 25 because test expression $4 == 9$ is false.

The conditional operator might be fascinating you but certainly one tip regarding ?:, I would like all of you to keep in mind and which is being told in form of following tip.

TIP

Beware that the conditional operator has a low precedence.

The conditional operator has a lower precedence than most other operators that may produce unexpected results sometimes. Consider the following code :

```
n = 500 ;
bonus = n + sales > 15000 ? 250 : 50 ;
```

The above code is trying to add n and the value 250 or 50 depending upon whether $\text{sales} > 15000$ is true or false. But this code will not work in the desired manner. The above code will be interpreted as follows :

```
bonus = (n + sales) > 15000 ? 250 : 50 ;
```

because operator '+' has higher precedence over `>` and `?`:

Therefore, for the desired behaviour the conditional expression should be enclosed in parentheses as shown below :

```
bonus = n + (sales > 15000 ? 250 : 50) ;
```

The conditional operator can be *nested* also i.e., any of the *expression2* or *expression3* can itself be another conditional operator. For example, the following expression first tests the condition if *class* ≥ 10 , if it is true it tests for another condition if *marks* ≥ 80 .

```
class >= 10 ? (marks >= 80 ? 'A' : 'B') : 'C' ;
```

The conditional operators can either be in the true part i.e., before colon `:` or/and in the false part i.e., after colon `(:)`.

5.7.8B The [] Operator

The square brackets are used to declare arrays, to create arrays, and to access a particular element in an array. Similar data items, such as *marks of 20 students* or *sales of 30 salesmen* etc., are combined together in the form of arrays. Here's an example of an array declaration :

```
float[ ] arrayOfFloats = new float[10] ;
```

The previous code declares an array that can hold ten floating point numbers. Here's how you would access the 7th item in that array :

```
arrayOfFloats[6] ;
```

Please note that first element of array is referred to as *array-name[0]* i.e., to refer to first item of array namely *arrayOfFloats*, we shall write *arrayOfFloats[0]*. We are not going into further details of arrays right now. Arrays shall be discussed in details in chapter 15.

5.7.8C The . Operator

The dot `.` operator accesses instance members of an object or class members of a class. You will learn more about this operator in Chapter 16.

5.7.8D The () Operator

When declaring or calling a method, the method's arguments are listed between parenthesis `(and)`. You can specify an empty argument list by using `()` with nothing between them. Chapter 13 *Functions* covers methods in details.

5.7.8E The (type) Operator

This operator casts (or "converts") a value to the specified type. You'll see the usage of this operator a little later in this chapter, under the topic *Type Conversion*.

5.7.8F The new Operator

You can use the *new* operator to create a new object or a new array. You'll find examples highlighting the usage of new operator in a later section - *Objects as instances of class* – in this chapter.

5.7.8G The instanceof Operator

The *instanceof* operator tests whether its first operand is an instance of its second.

op1 instanceof op2

op1 must be the *name-of-an-object* and *op2* must be the *name-of-a-class*. An object is considered to be an instance of a class if that object directly or indirectly descends from that class. For instance, following statement tests whether *goalKeeper* is an object (or instance) of class type *FootballPlayer*.

```
if goalKeeper instanceof FootballPlayer {  
    :  
}
```

5.7.9 Operator Precedence

Operator precedence determines the order in which expressions are evaluated. This, in some cases, can determine the overall value of the expression. For example, take the following expression :

y = 6 + 4/2

Depending on whether the *6 + 4* expression or the *4/2* expression is evaluated first, the value of *y* can end up being 5 or 8. Operator precedence determines the order in which expressions are evaluated, so you can predict the outcome of an expression. In general, increment and decrement expressions are evaluated before arithmetic expressions ; arithmetic expressions are evaluated before comparisons, and comparisons are evaluated before logical expressions. Assignment expressions are evaluated last.

Table 5.17 shows the specific precedence of the various operators in Java. Operators further up in the table are evaluated first ; operators on the same line have the same precedence and are evaluated left to right based on how they appear in the expression itself. For example, given that same expression

y = 6 + 4/2

you now know, according to this table, that division is evaluated before addition, so the value of *y* will be 8.

You always can change the order in which expressions are evaluated by using parentheses around the expressions you want to evaluate first. You can nest parentheses to make sure that expressions evaluate in the order you want them to (the innermost parenthetical expression is evaluated first). Consider the following expression :

y = (6 + 4)/2

This results in a value of 5, because the *6 + 4* expression is evaluated first, and then the result of that expression (10) is divided by 2.

Parentheses also can be useful in cases where the precedence of an expression isn't immediately clear. In other words, they can make your code easier to read. Adding parentheses doesn't hurt, so if they help you figure out how expressions are evaluated, go ahead and use them.

Table 5.17 Operator Precedence and Associativity

Operator	Notes	Associativity
. [] ()	Parentheses () are used to group expressions ; a dot (.) is used for access to methods and variables within objects and classes ; and [] is used for arrays	Left to Right
++ -- ! ~ instanceof	Returns true or false based on whether the object is an instance of the named class or any of that class's superclasses	Right to Left
new (type) expression	The new operator is used for creating new instances of classes ; () in this case is for casting a value to another type	Right to Left
* / %	Multiplication, division, modulus	Left to Right
+ -	Addition, subtraction	Right to Left
<< >> >>>	Bitwise left, right shift, and the zero fill right shift	Left to Right
< > <= >=	Relational comparison tests	Left to Right
== !=	Equality	Left to Right
&	AND	Left to Right
^	XOR	Left to Right
	OR	Left to Right
&&	Logical AND	Left to Right
	Logical OR	Left to Right
?:	Shorthand for if...then...else	Right to Left
= += -= *= /= %= ^=	Various assignments	Right to Left
&= = <<= >>= >>>=	Various assignments	Right to Left

Operator Associativity

There is a linked term – **Operator Associativity**. Associativity rules determine the grouping of operands and operators in an expression with more than one operator of the same precedence. When the operations in an expression all have the same precedence rating, the *associativity* rules determine the order of the operations. For most operators, the evaluation is done left to right, e.g.,

$$x = a + b - c ;$$

Here, addition and subtraction have the same precedence rating and so a and b are added and then from this sum c is subtracted. Again, parentheses can be used to overrule the default associativity, e.g.,

$x = a + (b - c);$

However, the assignment and unary operators, are associated right to left, e.g.,

$x += y -= -4;$

is equivalent to

$x += (y -= (-4));$

Table 5.17 also lists associativity of various operators in Java.

Example 5.5. Write a program that computes sum of three given numbers and finds the largest of the three.

Solution.

```
class Ex10_5 {
    public static void main ( String args[ ] ) {
        int a, b, c ;
        a = 3; b = 5; c = 1;
        int large, sum ;
        sum = a + b + c ;
        System.out.println ( "Sum of " + a + " and " + b + " and " + c + " is : " + sum ) ;
        large = ( ( a > b ) ? ( ( a > c ) ? a : c ) : ( ( b > c ) ? b : c ) );
        System.out.println ( "Largest of " + a + ", " + b + " & " + c + " is: " + large ) ;
    }
}
```

Sum of 3 and 5 and 1 is : 9

Largest of 3 , 5 & 1 is: 5

After this discussion of operators, let us move on to our next topic of discussion – the *expressions*.

5.8 Expressions

An expression is composed of one or more *operations*. The objects of the *operation(s)* are referred to as *operands*. The operations are represented by *operators*. Therefore, operators, constants, and variables are the constituents of expressions.

The expressions in Java can be of any type : arithmetic expression, relational (or logical) expression, compound expression etc. Let us discuss each of these expression types in details.

Type of operators used in an expression determine the expression type. For instance, if an expression is formed using arithmetic operators, it is an arithmetic expression ; if an expression has relational and/or boolean operators, it is a boolean expression. An arithmetic expression always results in a number (integer or real) and a logical expression always results in a logical value i.e., either true or false.

An **Expression** in Java is any valid combination of operators, constants, and variables i.e., a legal combination of Java tokens.

5.8.1 Arithmetic Expressions

Arithmetic expressions can either be **pure integer expressions** or **pure real expressions**. Sometimes a **mixed expression** can also be formed which is a mixture of real and integer expressions.

Integer expressions are formed by connecting integer constants and/or integer variables using integer arithmetic operators.

The following are valid integer expressions :

```
final int count = 30 ;
int I, J, K, X, Y, Z ;
```

- | | | | |
|------------------|-----------|--------------|--------------------------------|
| (a) I | (b) $-J$ | (c) $K - X$ | (d) $K + X - Y + \text{count}$ |
| (e) $-J + K * Y$ | (f) J/Z | (g) $Z \% X$ | |

Real expressions are formed by connecting real constants and/or real variables using real arithmetic operators.

The following are valid real expressions :

```
final float bal = 250.53f ;
float qty, amount value ;
double fin, inter ;
```

- | | | |
|---|---|--|
| (i) qty/amount | (ii) $\text{qty} * \text{value}$ | (iii) $(\text{amount} + \text{qty} * \text{value}) - \text{bal}$ |
| (iv) $\text{fin} + \text{qty} * \text{inter}$ | (v) $\text{inter} - (\text{qty} * \text{value}) + \text{fin}$ | |

Rule for these arithmetic expressions is the same and it states that :

An arithmetic expression may contain just one numeric variable or a constant, or it may have two or more variables or/and constants, or two or more expressions joined by valid arithmetic operators. Two or more variables or operators should not occur in continuation.

Apart from variables, constants and arithmetic operators, an arithmetic expression may consist of Java's mathematical functions that are part of Java standard library and are available through **Math** class defined in **java.lang** package.

Following Table 5.18 lists various math functions that are defined in the **Math** class of **Java.lang** package.

You can use these math functions as per following syntax :

Math.function_name(argument list)

The arguments are the values required by a function to work upon.

For example, to calculate a^b , you may write

Math.pow(a, b)

Note In **pure expressions**, all the operands are of same type. And in **mixed expressions**, the operands are of mixed or different data types.

Table 5.18 Math Functions Available through Math Class

Functions	Action
$\sin(x)$	This function returns the sine of the angle x in radians
$\cos(x)$	This function returns the cosine of the angle x in radians
$\tan(x)$	This function returns the tangent of the angle x in radians
$\text{asin}(y)$	This function returns the angle whose sine is y
$\text{acos}(y)$	This function returns the angle whose cosine is y
$\text{atan}(y)$	This function returns the angle whose tangent is y
$\text{atan2}(x, y)$	This function returns the angle whose tangent is x/y
$\text{pow}(x, y)$	This function returns x raised to y (x^y)
$\text{exp}(x)$	This function returns e raised to x (e^x)
$\log(x)$	This function returns the natural logarithm of x
$\text{sqrt}(x)$	This function returns the square root of x
$\text{ceil}(x)$	This function returns the smallest whole number greater than or equal to x. (Rounded up)
$\text{floor}(x)$	This function returns the largest whole number less than or equal to x (Rounded down)
$\text{rint}(x)$	This function returns the rounded value of x
$\text{abs}(a)$	This function returns the absolute value of a
$\text{max}(a, b)$	This function returns the maximum of a and b
$\text{min}(a, b)$	This function returns the minimum of a and b.

Following are examples of *valid arithmetic expressions*:

Given

```
int a, b, c ;    float, p, q, r ;    double x, y, z ;
```

- (i) a/b
- (ii) $p/q + a - c$
- (iii) $x/y + p * a/b$
- (iv) $(\text{Math.sqrt}(b) * a) - c$
- (v) $(\text{Math.ceil}(p) + a) / c$
- (vi) $\text{fmod}(c, b) + x/y - z/q + c$

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

Following are examples of *invalid arithmetic expressions*:

Given

```
int, a, b, c ;    float, p, q, r ;    double x, y, z ;
```

- (i) $x + * r$ two operators in continuation.
- (ii) $q(a + b - z / 4)$ operator missing between q and parenthesis.
- (iii) $\text{Math.pow}(0, -1)$ Domain error because if base = 0 then exp should not be $<= 0$.
- (iv) $n \log(-3) + p / q$ Domain error because logarithm of a negative number is not possible.

Example 5.6. Write the corresponding C++ expressions for the following mathematical expressions :

(i) $\sqrt{a^2 + b^2 + c^2}$

(ii) $2 - ye^{2y} + 4y$

(iii) $p + q / (r + s)^4$

(iv) $(\cos x / \tan^{-1} x) + x$

(v) $|e^x - x|$

Solution.

(i) `Math.sqrt (a * a + b * b + c * c)`

(ii) `2 - y * Math.exp(2 * y) + 4 * y`

(iii) `p + q / Math.pow((r + s), 4)`

(iv) `(Math.cos(x) / Math.atan(x)) + x`

(v) `Math.abs (Math.exp(x) - x)`

Type Conversion

When constants and variables of different types are mixed in an expression, they are converted to the same type.

Java facilitates the type conversion in two forms :

The process of converting one predefined type into another is called **Type Conversion**.

(i) Implicit type conversion

An implicit type conversion is a conversion performed by the compiler without programmer's intervention. An implicit conversion is applied generally whenever differing data types are intermixed in an expression (mixed mode expression), so as not to lose information.

The Java compiler converts all operands upto the type of the largest operand, which is called **type promotion**. This is done operation by operation, as described in the following type conversion rules :

- ◆ If either operand is of type **double**, the other is converted to **double**.
- ◆ Otherwise, if either operand is of type **float**, the other is converted to **float**.
- ◆ Otherwise, if either operand is of type **long**, the other is converted to **long**.
- ◆ Otherwise, both operands are converted to type **int**.

Once these conversion rules have been applied, each pair of operands is of the same type and the result of each operation is the same as the type of both operands. Consider the example in following figure (Fig. 5.4).

```
char ch ;      int i ;          float fl ;          double db ;      double ld ;  
result = (ch / i) + (fl * db) - (fl + i) + (fd / fi)
```

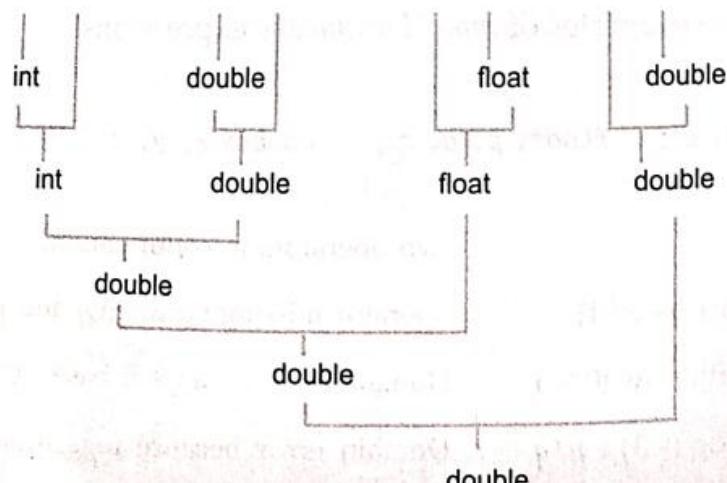


Figure 5.4 A type conversion example.

Although coercion exempts the user from worrying about different datatypes of operands, yet it has one disadvantage.

Coercions decrease the type error detection ability of the compiler.

The implicit type conversion wherein datatypes are promoted is known as **Coercion**.

(ii) Explicit type conversion

An explicit type conversion is user-defined that forces an expression to be of specific type.

Type casting in Java is done as shown below :

`(type) expression`

The explicit conversion of an operand to a specific type is called **Type Casting**.

where *type* is a valid Java data type to which the conversion is to be done. For example, to make sure that the expression $(x + y / 2)$ evaluates to type *float*, write it as :

`(float) (x + y / 2)`

casts are often considered as operators. As an operator, a cast is *unary* and has the same precedence as any other unary operator.

Below is a table that indicates to which of the other primitive types a given primitive data type can be cast. The symbol C indicates that an explicit cast is required since the precision is decreasing. The symbol A indicates that the precision is increasing so an automatic cast occurs without the need for an explicit cast. N indicates that the conversion is not allowed.

	<i>int</i>	<i>long</i>	<i>float</i>	<i>double</i>	<i>char</i>	<i>byte</i>	<i>short</i>	<i>boolean</i>
<i>int</i>	-	A	A*	A	C	C	C	N
<i>long</i>	C	-	A*	A*	C	C	C	N
<i>float</i>	C	C	-	A	C	C	C	N
<i>double</i>	C	C	C	-	C	C	C	N
<i>char</i>	A	A	A	A	-	C	C	N
<i>byte</i>	A	A	A	A	C	-	A	N
<i>short</i>	A	A	A	A	C	C	-	N
<i>boolean</i>	N	N	N	N	N	N	N	-

The * asterisk indicates that the least significant digits may be lost in the conversion even though the target type allows for bigger numbers. For example, a large value in an *int* type value that uses all 32 bits will lose some of the lower bits when converted to *float* since the exponent uses 8 bits of the 32 provided for *float* values.

Assigning a value to a type with a greater range (e.g., from *short* to *long*) poses no problem, however, assigning a value of larger data type to a smaller data type (e.g., from *double* to *float*) may result in losing some precision. There are some other similar potential conversion problems that are listed below in Table 5.19.

Table 5.19 Potential Conversion Problems

S.No.	Conversion	Potential Problems
1.	Bigger floating-point type to smaller floating-point type (e.g., double to float)	Loss of precision (significant figures). Original value may be out of range for target type, in which case result is undefined.
2.	Floating-point type to integer type	Loss of fractional part. Original value may be out of range for target type, in which case result is undefined.
3.	Bigger integer type to smaller integer type (e.g., long to short)	Original value may be out of range for target type. Typically, just resulting in loss of information.

Expression Evaluation

As mentioned earlier, expressions can either be **pure expressions** or **mixed expressions**. Pure expressions have all operands of same datatypes, contrary to mixed expressions that have operands of mixed datatypes.

Evaluating Pure Expressions

Pure expressions produce the result having the same datatype as that of its operands e.g.,

```
int a = 5, b = 2, c ;
a + b will produce result 7 of int type.
a/b will produce result 2 of int type.
```

Notice that it will not produce 2.5, it will produce 2.

Now, can you answer this question ?

What will be the result produced by 100/11 ?

- (a) 9 (b) 9.999999

Well, you guessed it right. It is indeed 9 because of pure division operation in 100/11.

In Java, when a mixed expression is evaluated, it is first divided into component sub-expressions upto the level of two operands and an operator. Then the type of sub-expression is decided keeping in mind general conversion rules. Using the results of sub-expressions, the next higher level of expression is evaluated and its type is determined. This process is continued till you get the final result of the expression.

Following example illustrates this.

Example 5.7. Evaluate the following Java expression :

```
int a, mb = 2, k = 4 ;
a = mb * 3 / 4 + k / 4 + 8 - mb + 5 / 8;
```

Note You cannot typecast a boolean type to another primitive type and vice versa. So, you cannot cast a primitive type to an object reference, or vice versa.

Solution.

$$\begin{aligned}
 a &= mb * 3 / 4 + k / 4 + 8 - mb + 5^8 \\
 &= (2 * 3) / 4 + 4 / 4 + 8 - 2 + 5^8 \\
 &= 6 / 4 + 1 + 8 - 2 + 5^8 \\
 &= 1 + 1 + 8 - 2 + 0 \quad (\because 6/4 \text{ evaluates to } 1.5 \text{ but the result is } 1 \text{ (an int)}) \\
 &\quad \text{because } 6/4 \text{ is a pure int expression as both the operands} \\
 &\quad \text{are of int types. Similarly, } 5^8 \text{ evaluates to } 0.625 \text{ but} \\
 &\quad \text{the result is } 0 \text{ since it is also an int expression).} \\
 &= 10 - 2 = 8.
 \end{aligned}$$

Boolean Expressions

The expressions that result into *false* or *true* are called boolean expressions. The Boolean expressions are combination of constants, variables and logical and relational operators. The rule for writing boolean expressions states :

The following are examples of some valid boolean expressions :

- | | |
|-------------------------------------|---------------------------------|
| (i) $x > y$ | (ii) $(y + z) \geq (x/z)$ |
| (iii) $(a + b) > c \&& (c + d) > a$ | (iv) $(y > x) \mid\mid (z < y)$ |
| (v) $x \mid\mid y \&\& z$ | (vi) (x) |
| (vii) $(-y)$ | (viii) $(x - y)$ |
| (ix) $(x > y) \&\& (!y < z)$ | (x) $x <= !y \&\& z$ |

Note A boolean expression may contain just one signed or unsigned variable or a constant, or it may have two or more variables or/and constants, or two or more expressions joined by valid relational and/or logical operators. Two or more variables or operators should not occur in continuation.

5.9 Java Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;) :

- ◆ Assignment expressions
- ◆ Any use of `++` or `--`
- ◆ Method calls
- ◆ Object creation expressions

These kinds of statements are called *expression statements*. Here are some examples of expression statements :

```

 aValue = 8933.234 ;           // assignment statement
 aValue++ ;                   // increment statement
 System.out.println(aValue) ;   // method call statement
 Integer integerObject = new Integer[4] ; // object creation statement
 
```

In addition to these kinds of expression statements, there are two other kinds of statements. A *declaration statement* declares a variable. You've seen many examples of declaration statements.

```

 double aValue = 8933.234 ;           // declaration statement
 
```

A *control flow statement* regulates the order in which statements get executed. The *for loop* and the *if* statement are both examples of control flow statements. You'll learn about these in chapters 5 and 6.

Block

A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following listing shows two blocks :

```
if (Character.isUpperCase(aChar))
{
    System.out.println("The character" + aChar + "is upper case.");
}
else
{
    System.out.println("The character" + aChar + "is lower case.");
    System.out.println("Thank You");
}
```

Let us make it more clear. The above code fragment is re-written to make you understand the blocks in java program :

```
if (Character.isUpperCase(aChar))

A block → {           // block1 begins
            System.out.println("The character" + aChar + "is upper case.");
            }           // end of block1

        else

Another block → {           // block2 begins
            System.out.println("The character" + aChar + "is lower case.");
            }           // end of block2
```

See, the beginning and end of blocks have been marked.

In this book, we shall be following conventional style where opening brace of the block is not put in a separate line, rather it is placed in continuation with the previous statement (whose part the block is). For instance, rather than showing

```
if (a > b)
{
    :
}
```

we shall be writing

```
if (a > b) {
```



*opening brace of the block in
continuation with previous
statement*

A Block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

Null or Empty Statement

As you know, in Java programs, statements are terminated with a semicolon ; . The simplest statement of them all is the empty, or null statement. It takes the following form :

 it is a null statement
;

A null statement is useful in those instances where the syntax of the language requires the presence of a statement but where the logic of the program does not. We will see it in loops and their bodies.

5.10 Significance of Classes

Okay, so now that you've understood the Java fundamentals it's time you learnt how object-oriented design is implemented in Java.

The basic unit of object-orientation in Java is *the class*. The class is often described as a *blueprint for an object*. It allows a programmer to define all of the properties and methods that internally define an object, all of the API (*Application programming Interface*) methods that externally define an object, and all of the syntax necessary for handling encapsulation, inheritance and polymorphism. Therefore, we can say that a class is the **BLUEPRINT** of an object. A class defines the types of shared characteristics, such as :

- ❖ The set of attributes
- ❖ The set of behaviour
- ❖ How to construct one
- ❖ How to destroy one (sometimes)

And the objects are "instances" of a class.

In its role as a *blueprint*, the class specifies what an actual object will look like. But it is not an object. You can think of a class as a *cookie cutter* and an instance as an *actual cookie*. Similarly, you can think of a class as a blueprint of a house and an instance as an actual house.

Class as Basis of all Computation

In Java, the class forms the basis of all computation. Anything that has to exist as a part of a Java program has to exist as a part of a class, whether that is a variable or a function or any other code-fragment. Unlike other OOP languages such as C++ that allow the existence of variables and functions outside any class. The reason being that Java is a pure Object Oriented Language. Here all functionality revolves around classes and object, as in real world. Therefore, if you want to use certain variables and functions in Java, you have to make them part of a class. All JAVA programs consist of **objects** (data and behaviour) that "interact" with each other by calling *methods*. All data is stored within objects which are instances of a class. See, without classes can be no objects and without objects, no computation can take place in Java. Thus, classes form the basis of all computation in Java.

Defining Classes

A Java program consists of objects, from various classes, interacting with one another. Before we go into the details of how you can define classes, let's review some of the general

properties of classes. A value of class type is called an **object**. An object is usually referred to as an **instance of the class** rather than as a value of the class, but it is a value of the class type. An object is a value of the class type much like a value, such as 5, of a primitive type, like int, is a value of a variable of that type. However, an object typically has multiple pieces of data and has **methods** (actions) it can take. Each object can have different data but all objects of the class have the same types of data and all objects in a class have the same methods. We generally say that data and methods belong to the object, and that is an acceptable point of view. The data certainly does belong to the object, but since all objects in a class have the same methods, it also would be correct to say that the methods actually belong to the class.

In this section we are going to learn about how to define classes in Java.

Now consider the code fragment shown below defines a class called City.

```
public class City { This class definition will get
    stored in a file named City.Java
    public String name ;      // variable name will be name of the City
    public long population ;  // will hold City's population
    public void display( )
    {
        System.out.println("City name :" + name) ;
        System.out.println("Population :" + population) ;
    }
}
```

Now let us examine this code line by line. The firstline

```
public class City
```

defines the name of the class which is City. The keyword class ensures that it is a **class** and the keyword **public** means it is available in entire program.

The brace following the public Class City

```
{
```

marks the beginning of class' block.

The next two lines

```
public String name ;
public long population ;
```

declare the data members of the class to define its characteristics. The keyword **public** simply means that there are no restrictions on how these data members (also called **instance variables**) are used.

Each of these lines declares one instance variable name. You can think of an object of the class as a complex item with instance variables inside of it. So, you can think of an instance variable as a smaller variable inside each object of the class. In this case, the instance variables are called **name** and **population**.

The next four lines

```
public void display( )
{
    System.out.println("City name :" + name);
    System.out.println("Population :" + population);
}
```

define a public method of the class which defines the behaviour of the class. The name of the method is `display()`. By looking at its code, you can easily make out what its functionality is like.

The last line

```
}
```

marks the end of the class' block.

Now how you can use this class, would become clear to you only after you go through the next section – **Objects as Instances of Classes**.

Rules to Declare a Class

In order to bring a class into existence in Java program, it should be declared. A class is declared using keyword `class`. The generic syntax for class declaration in Java is :

```
[<accessSpecifier>] [<modifier>] class <class_name>
{
    statements defining class come here
}
```

The angle-brackets `< >` mean these names are to be provided by the programmer and [] brackets mean, this part is optional.

The class name must be provided while declaring a class. This name is used to refer to the class whenever an instance (the object) of the class is created. The class name must be a legal identifier in Java.

While naming classes, generally nouns are used and the first letter is given in uppercase e.g., `City` or `Date` etc.

The curly brackets { } mark the beginning of a class and/or a method. The curly brackets are also used to specify block statements.

A class is a blueprint or prototype that you can use to create many objects. The implementation of a class is comprised of two components : the *class declaration* and the *class body*.

```
classDeclaration {
    // classBody
}
```

Note When a class is declared, no memory space is allotted to it. This is because a class is simply a blueprint or logical placeholder containing objects and methods. Memory space is allocated when objects of a class type are created.

The Class Declaration

The class declaration component declares the name of the class along with other attributes such as the class's superclass, and whether the class is public, final, or abstract.

The Class Body

The class body follows the class declaration and is embedded within curly braces '{' and '}'. The class body contains declarations for all *instance variables* and *class variables* (known collectively as *member variables*) for the class. In addition, the class body contains declarations and implementations for all instance methods and class methods (known collectively as *methods*) for the class.

The following template shows the form of a class definition that is most commonly used; however, it is legal to intermix the method definitions and the instance variable declarations.

```
public class Class_Name {
    Instance_Variable_Declaration_1
    Instance_Variable_Declaration_2
    ....
    Instance_Variable_Declaration_Last
    Method_Definition_1
    Method_Definition_2
    ....
    Method_Definition_Last
}
```

Declaring Member Variables

A class's state is represented by its member variables. You declare a class's member variables in the body of the class. Typically, you declare a class's variables before you declare its methods, although this is not required.

```
classDeclaration {
    member variable declarations
    method declarations
}
```

The class variables can be of *two types*:

- Class variable (static variable).** A data member that is declared once for a class. All objects of the class type, share these data members, as there is single copy of them available in memory.
- Instance variable.** A data member that is created for every object of the class. For example, if there are 10 objects of a class type, there would be 10 copies of instance variables, one each for an object.

Consider the following code :

```
public class Sample {
    int anInt ;                                // instance variable
    float aFloat ;                             // instance variable
    static float anotherFloat ;                // class variable
    :
}
```

Note

To declare variables that are members of a class, the declarations must be within the class body, but **not** within the body of a method. Variables declared within the body of a method are *local* to that method i.e., available and accessible only inside the method.

Suppose there are five objects created for class type **Sample**. Then there would be five copies of variables **aInt** and **aFloat** but there would be one copy of variable **anotherFloat** which all five objects can share. (Fig. 5.5)

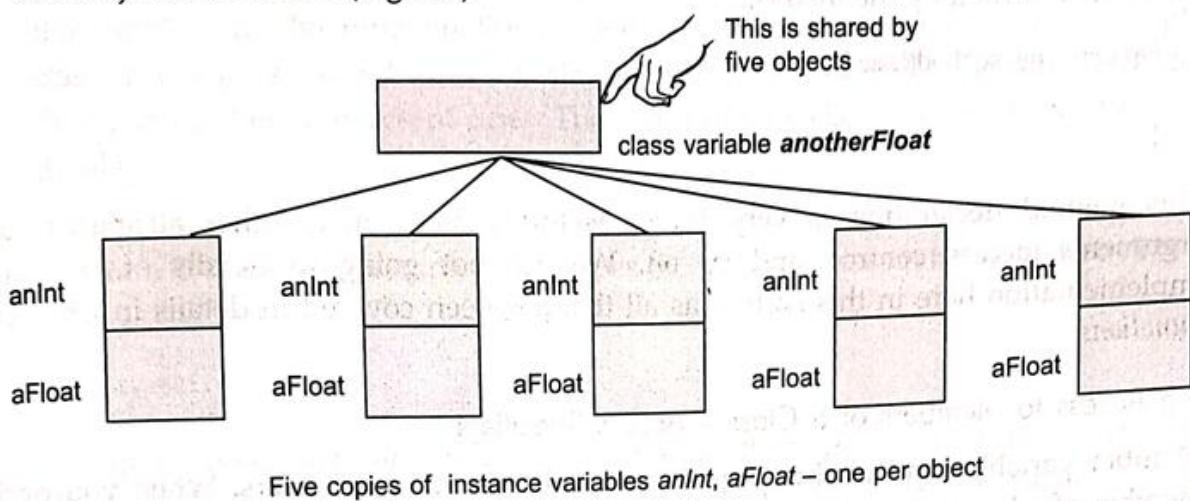


Figure 5.5 Class variable and Instance variables.

Notice that class variables are declared by adding a keyword **static** before the variable declaration.

TIP

Keyword **static** in the variable declaration makes it class variable.

Methods are similar : Your classes can have *instance methods* and *class methods*. *Instance methods* operate on the current object's instance variables but also have access to the *class variables*. *Class methods* (also called *static methods*), on the other hand, **cannot access** the *instance variables* declared within the class (unless they create a new object and access them through the object). Also, class methods can be invoked on the class, you don't need an instance to call a class method.

Defining Methods

As you know, objects have behaviour that is implemented by its methods. Other objects can ask an object to do something by invoking its methods. This section tells you everything you need to know about writing methods for your Java classes.

In Java, you define a class's methods in the body of the class for which the method implements some behaviour. Typically, you declare a class's methods after its variables in the class body although this is not required.

Implementing Methods

Similar to a class implementation, a method implementation consists of *two parts* : the method declaration and the method body

```
methodDeclaration {
    methodBody
}
```

The Method Declaration

At minimum, a method declaration has a name and a return type indicating the data type of the value returned by the method :

```
returnType methodName( ) {  
    ...  
}
```

This method declaration is very basic. Methods have many other attributes such as arguments, access control, and so on. We are not going in details of the methods implementation here in this section as all this has been covered in details in Chapter 12 – *Functions*.

Controlling Access to Members of a Class – Access Specifiers

Member variables and methods are known collectively as members. When you declare a member of a Java class, you can allow or disallow other objects of other types access to that member through the use of *access specifiers* or *access or visibility rules*.

Access or visibility rules determine whether a method or a data variable can be accessed by another method in another class or subclass.

We have used the public modifier frequently in class definitions. It makes classes, methods, or data available to any other method in any other class or subclass.

Java provides for *four* access modifiers :

1. *public* access by any other class anywhere.
2. *protected* accessible by the package classes and any subclasses that are in other packages .
3. *Default* accessible to classes in the same package but not by classes in other packages, even if these are subclasses. (But note that *default* is not a keyword ; in the absence of any other access specifier, default access takes place.)
4. *private* accessible only within the class. Even methods in subclasses in the same package do not have access.

These access rules allow one to control the degree of encapsulation of your classes. For example, you may distribute your class files to other users who can make subclasses of them.

By making some of your data and methods private, you can prevent the subclass code from interfering with the internal workings of your classes.

Also, if you distribute new versions of your classes at a later point, you can change or eliminate these private attributes and methods without worrying that these changes will affect the subclasses.

The functionality of these access specifiers would become more clear to you after you go through chapter 7 where we shall learn to use classes and objects in greater details.

Note Java file can have only one public class.

5.11 Objects as Instances of Class

A class defines only a blueprint and its concrete version comes into effect only through objects that implement the functionality as defined by class. Recall our class example of City class. The objects created from this class will have two variables : *name* and *population* ; and they will be able to represent cities. The object of City class will also have a method namely *display()*.

An object of a class is typically named by a variable of the class type. For example, the program CityTrial in Example 5.8 declares the two variables metro1 and metro2 to be of type City, as follows :

```
City metro1, metro2 ;
```

This gives us variables of the class City, but so far there are no objects of the class. Objects are class values that are named by the variables. To obtain an object you must use the new operator to create a "new" object. For example, the following creates an object of the class City and names it with the variable metro1 :

```
metro1 = new City( ) ;
```

We will discuss this kind of statement in more detail later in this chapter when we discuss something called a *constructor*. For now simply note that

```
Class_Variable = new Class_Name( ) ;
```

creates a new object of the specified class and associates it with the class type variable. Since the class variable now names an object of the class, we will often refer to the class variable as an object of the class. (This is really the same usage as when we refer to an int variable n as "the integer n", even though the integer is strictly speaking not n but the value of n.)

Unlike what we did in Example 5.7, the declaration of a class type variable and the creation of the object are more typically combined into one statement as follows :

```
City metro1 = new City( ) ;
```

THE new OPERATOR

The new operator is used to create an object of a class and associate the object with a variable that names it.

Syntax

```
Class_Variable = new Class_Name( ) ;
```

Example

```
City metro ;
metro = new City( ) ;
```

which is usually written in the following equivalent form

```
City metro = new City( ) ;
```

Note To instantiate an object, Java uses the keyword "new".

Example 5.8. Program to illustrate usage of classes and objects.

```

class City
{
    public class City
    {
        public String name; // variable name will be name of the City
        public long population; // will hold City's population
        public void display()
        {
            System.out.println("City name :" + name);
            System.out.println("Population :" + population);
        }
    }
}

1 public class CityTrial
2 {
3     public void CityTrialTest()
4     {
5         City metro1, metro2;
6         metro1 = new City();
7         metro2 = new City();
8         metro1.name = "Delhi";
9         metro1.population = 10000000;
10        System.out.println("City metro1's details :");
11        metro1.display();
12        display() method is being invoked for object metro1.
13        An object can invoke public method of its class type.
14        metro2.name = "Banglore";
15        metro2.population = 5000000;
16        System.out.println("City metro2's details :");
17        metro2.display();
18    }
}

```

The output produced by above code will be as follows :

```

City metro1's details :
City Name :Delhi
Population :10000000
City metro2's details :
City Name :Banglore
Population :5000000

```

Note

Testing/Using a class. In order to use and test a class (say sample class) and its members, create another class say A (or any name you like), add **main method** in it and then use the class (sample class here) and methods to be tested inside it. Make sure to store this class in file having same name as that of class and extension .java. For example, if the class name is A that contains **main method**, the file name should be A.java.

Instance Variables and Methods

Let us understand the details about instance variables using the class and program in Example 5.8. Each object of the class **City** has two instance variables, which can be named by giving the object name followed by a dot and the name of the instance variable. For example, the object **metro1** in the program **CityTrial** has the following two instance variables :

```
metro1.name  
metro1.population
```

Similarly, if you replace **metro1** by **metro2**, you obtain the two instance variables for the object **metro2**. Note that **metro1** and **metro2** together have a total of four instance variables.

The instance variables **metro1.name** and **metro2.name**, for example, are two different (instance) variables.

The instance variables in Example 5.8 can be used just like any other variable. For example, **metro1.name** can be used just like any other variable of type **String**. The instance variable **metro1.population** can be used just like any other variable of type **long**. Thus, although the following is not in the spirit of the class definition, it is legal and would compile :

```
metro1.name = "Hello friend." ;
```

More likely assignments to instance variables are given in the program **CityTrial**. The class **City** has only one method, which is named **display**.

An instance is an executable copy of a class. Another name for instance is object. There can be any number of objects of a given class in memory at any one time. Above discussion makes it clear that objects are the executable copy of the class, as a class cannot be directly executable. Hence, objects are said to be instances of classes.

With this we have come to the end of this chapter. Before moving on to solving some sample questions, let us revise whatever we have learnt so far.

Let Us Revise

- ❖ The operators are symbols/words (sometimes) that trigger some operations. The objects of operations are referred to as operands.
- ❖ Java provides arithmetic operators (+, -, *, /, %), increment/decrement operators (++, --), relational operators (>, <, >=, <=, ==, !=), logical operators (&&, ||, !), shift operators (>>, <<, >>>), bitwise operators (&, |, ^, ~), assignment operator (=) and other operators (?:, [], .. (parameters), (type), new, instanceof).
- ❖ Java expression is a legal combination of Java tokens.
- ❖ Pure expressions are those wherein all operands are of same datatype.
- ❖ Mixed expressions are those wherein operands are of different datatype.
- ❖ The process of converting one predefined type into another is called type conversion.
- ❖ Implicit type conversion is performed by Java compiler wherein smaller datatypes are promoted into higher datatypes. This process is also known as coercion.
- ❖ Explicit type conversion is performed by the programmer using (type) operator. This process is known as type casting.
- ❖ A block is a group of zero or more statements between balanced braces.
- ❖ An empty or null statement is semicolon (;) only. It is used when syntax demands a statement but the logic does not.

- ❖ A class is defined with keyword **class**.
- ❖ The members defined with keyword **static** become class members i.e., only one copy of them is created. All other members are instance members, which are created for every object of that class type.
- ❖ Object is an executable of class's functionality, hence it is called instance of the class.



Solved Problems

- Distinguish between a unary, a binary and a ternary operator. Give examples of Java operators for each one of them.

Solution. A unary operator requires a single operand. **unary +, unary -, ++, --, sizeof etc.** are some unary operators in Java.

A binary operator requires two operands.

+ (add), - (subtract), *, /, % etc. are some binary operators in Java.

A ternary operator requires three operands. ?: (the conditional operator) is a ternary operator in Java.

- Given the following code fragment

```
int ch = 20 ;
System.out.println (++ch) ;
System.out.println (ch) ;
```

(i) What output does the above code fragment produce ?

(ii) What is the effect of replacing `++ ch` with `ch + 1` ?

Solution.

(i) 21

21

(ii) `++ ch` not only replaces itself with value `ch + 1` i.e., 21 but also increments the value of `ch` i.e., after `++ ch` the value of `ch` is 21. Whereas `ch + 1` will only print the incremented value i.e., $20 + 1 = 21$; it will not increment the value of `ch`. Therefore, after replacing `++ ch` with `ch + 1`, the output of the program will be :

21

20

- What will be the result of following two expressions if `i = 10` initially ? (i) `++i <= 10` (ii) `i++ <= 10`

Solution. (i) false (ii) true

- Given the two following expressions :

(a) `val = 3` (b) `val == 3`

(i) How are these two different ?

(ii) What will be the result of the two if the value of `val` is 5 initially ?

Solution.

(i) The expression

(a) is an assignment expression and the expression

(b) is a relational expression that tests for equality.

(ii) The result of

(a) will be `val` having value 3 i.e., 3 and the result of

(b) will be boolean `false` because 5 is not equal to 3.

5. Construct an expression that is equal to the absolute value of a variable. That is, if a variable p is positive, the value of the expression is just p , but if p is negative, the value of the expression is $-p$, which would be positive.

Do it in two ways :

(i) using a mathematical function

(ii) using a conditional operator. (Do not use the mathematical function here).

Solution. (i) `Math.abs(p)` (ii) `p > 0 ? p : -p`

6. What output will the following code fragment produce ?

```
int val, res, n = 1000 ;
res = n + val > 1750 ? 400 : 200 ;
System.out.println(res) ;
```

(i) if the input is 2000 (ii) if the input is 1000 (iii) if the input is 500.

Solution.

- (i) 400 because the arithmetic operator `+` has higher precedence than `?` : operator thus the condition before `?` is taken as $(n + val)$ and $(1000 + 2000) > 1750$ is true.
(ii) 400 the reason is the same as explained above ($(1000 + 1000) > 1750$ is true).
(iii) 200 because $(1000 + 500) > 1750$ is false.

7. Given the following set of identifiers :

```
byte b ;
char ch ;
short sh ;
int intval ;
Long longval ;
float fl ;
```

Identify the datatype of the following expressions :

- (a) `'a' - 3` (b) `intval * longval - ch` (c) `fl + longval/sh`

Solution.

(a) int because

```
'a' - 3
|   |
char int
  [---]
    int
```

(c) float because

```
fl + (longval / sh)
|   |   |
float long short
  [---]
    long
      [---]
        float
```

(b) long because

```
(intval * longval) - ch
|   |   |
int long char
  [---]
    long
      [---]
        long
```

8. Using the declarations of Q. 7, identify which of the following assignments are unsafe (where there may be loss of data) :
 (a) `sh = intval` (b) `intval = longval` (c) `sh = b`

Solution.

- (a) unsafe because if sizes of `short` and `int` differ, there may be loss of data.
 (b) unsafe because if sizes of `int` and `long` differ, there may be loss of data.
 (c) safe, `short` is always larger than `byte`.

9. Suppose `x1` and `x2` are two double type variables that you want to add as integers and assign to an integer variable. Construct a Java statement for doing so.

Solution. Assuming that target variable is `res` of type `int`.

`res = (int) (x1 + x2);`

10. What is casting, when do we need it ?

Solution. Casting is a form of conversion, which uses the cast operator to specify by a type name in parentheses and is placed in front of the value to be converted. For example :

`result = (float) total / count ;`

They are helpful in situations where we temporarily need to treat a value as another type.

11. State the rules of operator precedence.

Solution. All expressions are evaluated according to an operator precedence hierarchy that establishes the rules that govern the order in which operations are evaluated.

Operators (`type`). `*`, `/`, and the remainder operator `%` are performed before `+` and `-`

Any expression in parentheses is evaluated first

The assignment operator has a lower precedence than any of the arithmetic operators.

12. We have two variables `X` and `Y`. Write Java statements to calculate the result of division of `Y` by `X` and calculate the remainder of the division.

Solution. `Y % X`

13. Will the value of `Y` be the same for the two cases given below ?

(i) `Y = ++X ;` (ii) `Y = X++ ;`

(Given the value of `X` is 42)

Solution. No.

14. What is a type, as this term relates to programming ?

Solution. A `type` or `datatype` represents a set of possible values. When we specify that a variable has a certain type, we are saying what values it can hold. When we say that an expression is of a certain type, you are saying what values the expression can have. For example, to say that a variable is of type `int` says that integer values in a certain range can be stored in that variable.

15. One of the primitive types in Java is `boolean`. What is the `boolean` type ? Where are `boolean` values used ? What are its possible values ?

Solution. The only values of type `boolean` are `true` and `false`. Expressions of type `boolean` are used in places where `true/false` values are expected.

16. What is a literal ?

Solution. A literal is a sequence of characters used in a program to represent a constant value. For example, '`A`' is a literal that represents the value `A`, of type `char`, and `17L` is a literal that represents the number `17` as a value of type `long`. A literal is a way of writing a value, and should not be confused with the value itself.

17. What does the computer do when it executes a variable declaration statement? Give an example.

Solution. A variable is a *box*, or *location*, in the computer's memory that has a name. The box holds a value of some specified type. A variable declaration statement is a statement such as

```
int x;
```

which creates the variable x. When the computer executes a variable declaration, it creates the box in memory and associates a name (in this case, x) with that box. Later in the program, that variable can be referred to by name.

18. Consider the following code snippet:

```
arrayOfInts[ j ] > arrayOfInts[ j+1 ]
```

What operators does the code contain?

Solution. [], >, [], +

19. Consider the following code snippet:

```
int i = 10;
int n = i++%5;
```

What are the values of i and n after the code is executed?

Solution. i is 11, and n is 0.

20. What are the final values of i and n if instead of using the postfix increment operator (i++), you use the prefix version (++i)?

Solution. i is 11, and n is 1.

21. What is the value of i after the following code snippet executes?

```
int i = 8;
i >>= 2;
```

Solution. i is 2.

22. What is the value of i after the following code snippet executes?

```
int i = 17;
i >>= 1;
```

Solution. i is 8.

23. What output shall be produced by following code?

```
class sp10_23 {
    public static void main(String[ ] args)
    {
        int x = 0;
        int y = 0;
        ++x;
        System.out.println("x=0; ++x; \t\t\t -> " + x);
        x = 0;           x++;
        System.out.println("x=0; x++; \t\t\t -> " + x);
        x = 0;           --x;
        System.out.println("x=0; --x; \t\t\t -> " + x);
        x = 0;           x--;
        System.out.println("x=0; x-- ; \t\t\t -> " + x);
        x = 0;           y = 0;           y = ++x;
        System.out.println("x = 0; y = 0; y = ++x; \t\t\t -> " + "y = " + y + " x = " + x );
        x = 0;           y = 0;           y = x++;
    }
}
```

```

System.out.println("x=0; y=0; y = x++; \t\t -> " + "y = " + y + " x = " + x);
    x = 0;          x = ++x;
System.out.println("x=0; x = ++x; \t\t\t -> " + x );
    x = 0;      x = x++;
System.out.println("x=0; x = x++; \t\t\t -> " + x );
    char c = 'a';      c++;
System.out.println("char c = 'a'; c++; \t\t\t -> " + c);
    --c;
System.out.println("char c = 'b'; --c; \t\t\t -> " + c);
    byte b = 2;
    byte b1;
System.out.println("byte b = 2; ++b; \t\t\t -> " + ++b);
    b1 = ++b;
System.out.println("byte b1 = 3; b1 = ++b; \t\t\t -> " + b1);
    b = 127;      b1 = ++b;
System.out.println("byte b1 = 127; b1 = ++b; \t\t -> " + b1);
    b = -128;      b1 = -- b;
System.out.println("byte b1 = -128; b1 = -- b; \t\t -> " + b1);
}
}

```

Solution.

x = 0; ++x;	-> 1
x = 0; x++;	-> 1
x = 0; -- x;	-> -1
x = 0; x --;	-> -1
x = 0; y=0; y = ++x;	-> y = 1 x = 1
x = 0; y=0; y = x++;	-> y = 0 x = 1
x = 0; x = ++x;	-> 1
x = 0; x = x++;	-> 0
char c = 'a'; c++;	-> b
char c = 'b'; -- c ;	-> a
byte b = 2; ++b;	-> 3
byte b1 = 3; b1 = ++b;	-> 4
byte b1 = 127; b1 = ++b;	-> -128
byte b1 = -128; b1 = -- b;	-> 127

24. Predict the output of following code ?

```

class sp10_24 {

    public static void main(String[ ] args) {
        int i = 0x10;           // decimal 16
        long j = 0x10L;         // decimal 16
        System.out.println( );
        System.out.println("\t i << 34 \t\t -> " + (i << 34) );
        System.out.println("\t i >> 34 \t\t -> " + (i >> 34) );
        System.out.println("\t i << -29 \t\t -> " + (i << -29) );
        System.out.println("\t i << 3 \t\t -> " + (i << 3) );
        System.out.println( );
    }
}

```

```

        System.out.println("\t l << 67 \t\t -> " + (i << 67) );
        System.out.println("\t l << -61 \t\t -> " + (i << -61) );
        System.out.println("\t l << 3 \t\t -> " + (i << 3) );
        System.out.println("\t l >> 67 \t\t -> " + (i >> 67) );
        System.out.println("\t l >> -61 \t\t -> " + (i >> -61) );
        System.out.println("\t l >> 3 \t\t -> " + (i >> 3) );
        System.out.println( );
        System.out.println("\t63 & 252 \t\t -> " + (63&252));
        System.out.println("\t63 | 252 \t\t -> " + (63|252));
        System.out.println("\t63 ^ 252 \t\t -> " + (63^252));
    }
}

```

Solution.

i << 34	-> 64
i >> 34	-> 4
i << -29	-> 128
i << 3	-> 128
l << 67	-> 128
l << -61	-> 128
l << 3	-> 128
l >> 67	-> 2
l >> -61	-> 2
l >> 3	-> 2
63 & 252	-> 60
63 252	-> 255
63 ^ 252	-> 195

25. Write a program to convert a temperature 122°F into Celcius and 100°C into Fahrenheit.

Solution.

```

class sp10_25
{
    public static void main (String a[ ])
    {
        double tempF,tempC;
        tempF = 122.0; // Temperature in Fahrenheit
        tempC = (tempF - 32 ) / 1.8 ;
        System.out.println(tempF + " degree Fahrenheit is equivalent to "
                           + tempC + " degree Celsius " );
        tempC = 100.0; // Temperature in Celsius
        tempF = (tempC * 1.8 ) + 32 ;
        System.out.println(tempC + " degree Celsius is equivalent to "
                           + tempF + " degree Fahrenheit" );
    }
}

```

Glossary

Arithmetic Operators Operators that carry out arithmetic calculations.

Constant A data item that never changes its value during a program run.

Explicit Type Conversion User-defined type conversion.

Expression Any valid combination of operators, constants, and variables.

Identifier Name given by user for a part of the program.

Implicit Type Conversion Type conversion automatically carried out by the compiler.

Integral Promotion Conversion of shorter integral types into bigger integral types.

Java Shorthand A way of combining an arithmetic operator and an assignment operator.

Keyword Reserved word having special meaning and purpose.

Literal Constant

Operator Symbol or keyword that represents a specific task.

String Literal Sequence of characters enclosed in double quotes.

Token The smallest individual unit in a program.

Type Casting Explicit conversion of an operand to a specific type.

Type Conversion The process of converting one predefined type into another.

Type Promotion Conversion of all operands upto the type of the largest operand.

Assignments

TYPE A : VERY SHORT/SHORT ANSWER QUESTIONS

1. Name the character set supported by Java.
2. What is meant by token ? Name the tokens available in Java.
3. What are keywords ? Can keywords be used as identifiers ?
4. What is an identifier ? What is the identifier forming rule of Java ?
5. Is Java case sensitive ? What is meant by the term 'case sensitive' ?
6. Which of the following are valid identifiers and why/why not :
 - Data_rec, _data, 1 data, data 1, my.file, asm, switch, goto, break ?
7. What are literals ? How many types of integer literals are available in Java ?
8. What is an integer constant ? Write integer forming rule of Java.
9. How many types of integer constants are allowed in Java ? How are they written ?
10. What kind of program elements are the following : 13, 'a', 4.38925, "a", main () ?
11. What kind of constants are the following : 14, 011, 0X2A, 17, 014, 0XBC1 ?
12. What is a character constant in Java ? How are nongraphic characters represented in Java ?
13. Why are characters \, ', " and ? typed using escape sequences ?
14. Which escape sequences represent the newline character & null character ?
15. What is meant by a floating constant in Java ? How many ways can a floating constant be represented into ?
16. Write the following real constants into exponent form : 23.197, 7.214, 0.00005, 0.319 .

17. Write the following real constants into fractional form :
 0.13E04, 0.417E-04, 0.4E-05, 0.123E02
18. What is the function of operators ? What are binary operators ? Give examples of arithmetic binary operators.
19. What will be the result of $a = 5 / 3$ if a is (i) float (ii) int ?
20. The expression $8 \% 3$ evaluates to _____.
21. Assuming that res starts with the value 25, what will the following code fragment print out ?

```
System.out.println (res - -) ;
System.out.println ( + + res) ;
```

22. What will be the value of $j = --k + 2 * k + (l = k, l++)$ if k is 20 initially ?
23. What will be the value of $P = P * ++J$ where J is 22 and $P=3$ initially ?
24. What will be the value of following, if $j = 5$ initially ?
 (i) $(5 * ++j) \% 6$ (ii) $(5 * j++) \% 6$
25. Write a statement that uses a conditional operator to set $grant$ to 10 if speed is more than 68, and to 0 otherwise.
26. What will be the result of following expression if (i) age = 25 (ii) age = 65 (iii) age = 85 ?
 $age > 65 ? 350 : 100$.
27. What will be the result of the following expression if
 (i) $ans = 700, val = 300$ (ii) $ans = 800, val = 700$?
 $ans - val < 500 ? 150 : 50$

28. Write an equivalent Java expressions for the following expressions :

$$(i) ut + \frac{1}{2} ft^2. \quad (ii) \sqrt{\sin a + \tan^{-1} a - e^{2x}} \quad (iii) |a| + b \geq |b| + a$$

$$(iv) \left(\frac{3x + 5y}{5x + 3y} - \frac{8xy}{2yx} \right)^{3/2} \quad (v) e^{|2x^2 - 4x|}$$

29. What is meant by implicit and explicit type conversion ?
30. What do you mean by type casting ? What is type cast operator ?
31. What will be the resultant type of the following expression if bh represents a byte variable, i is an int variable, fl is a float variable and db is a double variable ?
 $bh - i + db / fl - i * fl + db / i.$
32. What will be the resultant type of the following expression if fl is a float variable and db is a double variable ?
 $(int)(fl + db)$
33. Which class is used for using different mathematical functions in Java program ?
34. What are instance variables ? What are class variables ?
35. The modulus operator (%) can be used only with integer operands. True/False ?
36. All the bitwise operators have the same level of precedence in Java. True/False ?
37. When x is a positive number, the operation $x >> 2$ and $x >>> 2$ both produce the same result. True/False ?
38. In evaluating a logical expression of type
 $\text{boolean expression1} \&\& \text{boolean expression2}$
 both the boolean expressions are not always evaluated. True/False ?

39. The range of values for the long type data is
 (a) -2^{31} to $2^{31} - 1$ (b) -2^{64} to 2^{64} (c) -2^{63} to $2^{63} - 1$ (d) -2^{32} to $2^{32} - 1$
40. Which of the following represent(s) a hexadecimal number ?
 (a) 570 (b) (hex) 5 (c) 0X9G (d) 0X5
41. Which of the following assignments are invalid ?
 (a) float x = 123.4 (b) long m = 023 (c) int n = (int) false ; (d) double y = 0X756.
42. The default value of char type variable is
 (a) '\u0020' (b) '\u00ff' (c) " " (d) '\u0000'
43. What will be the result of the expression $13 \& 25$? ($13_{10} = 00001101_2$, $25_{10} = 00011001_2$)
 (a) 38 (b) 25 (c) 9 (d) 12
44. What will be the result of the expression $9 | 9$?
 (a) 1 (b) 18 (c) 9 (d) None of the above
45. Which of the following are correct ?
 (a) int a = 16, a >> 2 = 4 (b) int b = -8, b >> 1 = -4
 (c) int a = 16, a >>> 2 = 4 (d) All of the above
46. Which of the following will produce a value of 22 if $x = 22.9$?
 (a) ceil(x) (b) round(x) (c) abs(x) (d) floor(x)
47. Which of the following will produce a value of 10 if $x = 9.7$?
 (a) floor(x) (b) abs(x) (c) round(x) (d) ceil(x)
48. Given that :
`int x, m = 2000 ;
short y ;
byte b1 = - 40, b2 ;
long n ;`
 Which of the following assignment statements will evaluate correctly ?
 (a) $x = m * b1$; (b) $y = m * b1$; (c) $n = m * 3L$; (d) $x = m * 3L$;
49. Given the declarations
`boolean b ;
short x1 = 100, x2 = 200, x3 = 300 ;`
 Which of the following statements are evaluated to true ?
 (a) $b = x1 * 2 == x2$;
 (b) $b = x1 + x2 != 3 * x1$;
 (c) $b = (x3 - 2 * x2 < 0 || ((x3 = 400) < 2 ** x2))$;
 (d) $b = (x3 - 2 * x2 > 0) || ((x3 = 400)) 2 * x2$;
50. In which of the following code fragments, the variable x is evaluated to 8 ?
 (a) $\text{int } x = 32 ; x = x >> 33 ;$
 (b) $\text{int } x = 33 ; x = x >> 2 ;$
 (c) $\text{int } x = 35 ; x = x >> 2 ;$
 (d) $\text{int } x = 16 ; x = x >> 1 ;$

TYPE B : LONG ANSWER QUESTIONS

1. How are keywords different from identifiers ?
2. What are literals in Java ? How many types of literals are allowed in Java ?
3. How are integer constants represented in Java ? Explain with examples.
4. Can nongraphic characters be used and processed in Java ? How ? Give examples.
5. What are operators ? What is their function ? Give examples of some unary and binary operators.
6. What is the function of increment/decrement operators ? How many varieties do they come in ? How are these two varieties different from one another ?
7. What are unary, binary and ternary operators ? Give examples.
8. What are the purposes of following mathematical functions in Java ?
 - (i) atan()
 - (ii) atan2()
 - (iii) ceil()
 - (iv) exp()
9. What is meant by type conversion ? How is implicit conversion different from explicit conversion ?
10. Write arithmetic type conversion rules for float types and int types.
11. Determine the data type of the expression

$$(i) \left(\frac{100(1-pq)}{(q+r)} \right) - \left(\frac{(p+r)/s}{(\text{long})(s+p)} \right)$$

$$(ii) \left(\frac{2x+3y}{5w+6z} + \frac{8t}{5u} \right)^4$$

If p , x is an *int*, r , w is a *float*, q , y is a *long* and s , z is *double*, t is *short* and u is *long double*.

12. Evaluate the following Java expressions where a , b , c are integers and d , f are floating point numbers. The value of $a = 5$, $b = 3$ and $d = 1.5$.

- | | |
|-------------------------|-------------------------|
| (a) $f = a + b / a$ | (b) $c = d * a + b$ |
| (c) $c = (a++) * d + a$ | (d) $f = (++b) * b - a$ |

13. Write the Java equivalent expressions for the following :

- (a) $\text{volume} = 3.1459 r^2 h / 3$
- (b) $F_n = 0.5$ if $x = 30$
- (c) $fn = 0.9$ if $x = 60$

14. Suppose A , B , C are integer variables $A = 3$, $B = 3$, $C = -5$ and X , Y , Z are floating point variables where $X = 88$, $Y = 3.5$, $Z = -52$. Determine the value of the following expressions :

- (a) $A \% C$
- (b) $A * B / C$
- (c) $(A * C) \% B$
- (d) X / Y
- (e) $X / (X + Y)$
- (f) $\text{int}(X) \% \text{int}(Y)$

15. What are the potential problems in type conversions ? Discuss the benefits and loopholes of type casting.
16. Why do you think type compatibility is required in assigning values ?
17. How does a class form the basis of all computation ?
18. Distinguish between static variables and member variables (instance variables).

19. What are static functions ?

20. What will be the output of the following code ?

(a) `byte x = 64, y ;
y = (byte) (x << 2) ;
System.out.println(y) ;`

(b) (i) `00110011 & 11110000`
(ii) `00110011 ^ 11110000`
(iii) `00110011 | 11110000`

21. What will be the output of the following code :

(i) `byte b ;
double d = 417.35 ;
b = (byte) d ;
System.out.println(b) ;`

(ii) `int x = 10 ;
int y = 20 ;
if((x < y) || (x = 5) > 10)
 System.out.println(x) ;
else
 System.out.println(y) ;`

CHAPTER

6

Flow of Control

6.1 Introduction

Generally a program executes its statements from beginning to end. But not many programs execute all their statements in strict order from beginning to end. Most programs decide what to do in response to changing circumstances. These programs not only store data but they also manipulate data in terms of consolidating, rearranging, modifying data. To perform their manipulative miracles, programs need tools for performing repetitive actions and for making decisions. Java of course provides such tools by providing statements to attain so. Such statements are called **program flow control statements**.

This chapter is going to discuss various program flow control statements viz. *selection statements*, *iteration statements* and *jump statements*.

Every programming language provides some constructs which are necessary for it to be called as a programming language. Let us talk about these programming constructs briefly, before talking about selection statements.

6.2 Programming Constructs

In a program, statements may be executed *sequentially*, *selectively* or *iteratively*. Every programming language provides constructs to support *sequence*, *selection* or *iteration*. Let us discuss what is meant by sequence, selection or iteration constructs.

In This Chapter

- 6.1 Introduction
- 6.2 Programming Constructs
- 6.3 Selection Statements
- 6.4 Iteration Statements
- 6.5 Elements that Control a Loop (Parts of a Loop)
- 6.6 The for Loop — Fixed Number of Iterations
- 6.7 The while Loop
- 6.8 The do-while Loop
- 6.9 Nested Loops
- 6.10 Comparison of Loops
- 6.11 Jump Statements

Sequence. The *sequence construct* means the statements are being executed sequentially. This represents the default flow of statement (see Fig 6.1).

Every Java function execution begins with its first statement. Each statement in turn is executed (*sequence construct*). When the final statement of the function is executed, the function is done. This construct specifies the normal flow of control in a program and is the simplest one.

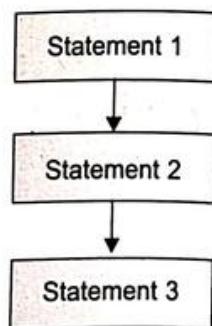


Figure 6.1 The Sequence construct.

Selection. The *selection construct* means the execution of statement(s) depending upon a *condition-test*. If a condition evaluates to *true*, a course-of-action (a set of statements) is followed *otherwise* another course-of-action (a different set of statements) if followed. This construct (selection construct) is also called *decision construct* because it helps in making decision about which set-of-statements is to be executed. Following figure (Fig. 6.2) explains selection construct.

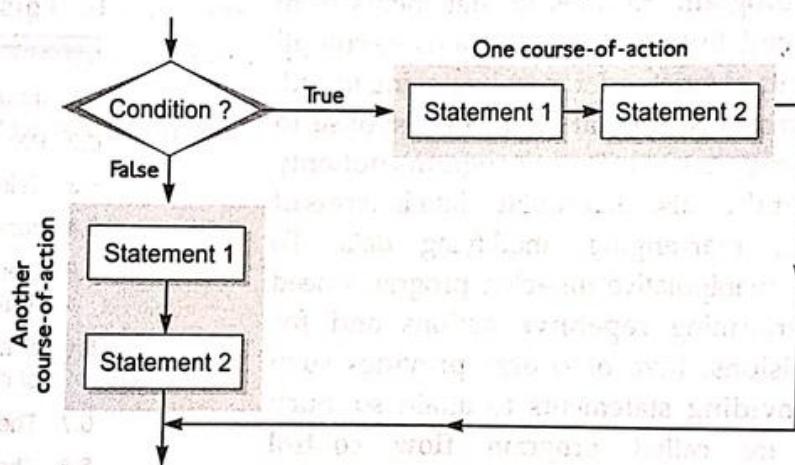


Figure 6.2 The Selection construct.

Iteration. The *iteration construct* means repetition of a set-of-statements depending upon a condition-test. Till the time a condition is *true* (or *false* depending upon the loop), a set-of-statements are repeated again and again. As soon as the condition becomes *false* (or *true* depending upon the statement), the repetition stops. The iteration construct is also called *looping construct*. Following figure (Fig. 6.3) illustrates an iteration construct.

The set-of-statements that are repeated again and again is called the *body of the loop*. The condition on which the *execution* or *exit* of the loop depends is called the *exit condition* or *test-condition*.

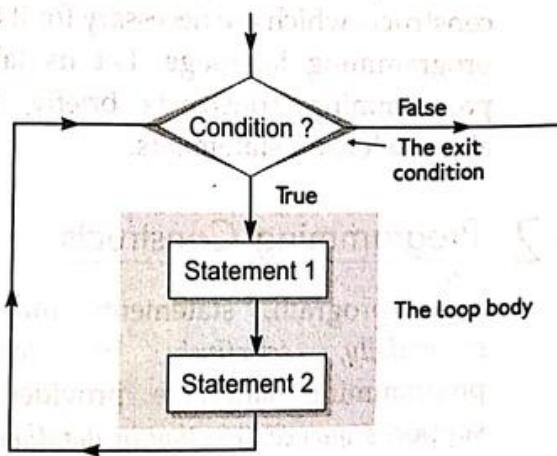


Figure 6.3 The Iteration construct.

Every programming language must provide all these constructs as the sequential program execution (the default mode) is inadequate to the problems we must solve. Java also provides statements that support these constructs. Coming section is going to discuss selection statements in Java.

Note Selection and Iteration constructs depend upon a conditional expression that determines what course-of-action is to be taken. A conditional expression evaluates to either true or false.

6.3 Selection Statements

The selection statements allow to choose the set-of-instructions for execution depending upon an expression's truth value. Java provides two types of selection statements : if and switch. In addition, in certain circumstances ?: operator can be used as an alternative to if statement. The selection statements are also called *conditional statements* or *decision statements*.

6.3.1 The if Statement of Java

An if statement tests a particular condition ; if the condition evaluates to *true*, a course-of-action is followed i.e., a statement or set-of-statements is executed. Otherwise (if the condition evaluates to *false*), the course-of-action is ignored. The syntax (general form) of the if statement is as shown below :

```
if (expression)
    statement ;
```

where a *statement* may consist of a single statement, a compound statement, or nothing (in case of empty statement). The *expression* must be enclosed in parentheses. If the *expression* evaluates to *true* i.e., a nonzero value, the *statement* is executed, otherwise ignored. For instance, the following code fragment :

```
if ( ch == ' ' )
    spaces ++ ;
```

checks whether the character variable *ch* stores a space or not ; if it does, the number of spaces are incremented by 1. Consider another example illustrating the use of if statement :

```
char ch ;
if (ch == ' ')
    System.out.println ("It is a space character") ;
if ( ch >= '0' & & ch <= '9' )
    System.out.println ("It is a digit") ;
```

The above example reads a character. If the character input is a space, it flashes a message specifying it. If the character input is a digit, it flashes a message specifying the same. The following example also makes use of an if statement.

```
if ( A > 10 & & B < 15 )
{
    C = (A - B) * (A + B) ;
    System.out.println ("The result is" + C) ;
}
```

The above example uses a compound statement (or a block) in the body of if.

All the examples of *if* you have seen so far allow you to execute a set of statements if a condition or expression evaluates to *true*. What if there is another course of action to be followed if the expression evaluates to *false*. There is another form of *if* that allows for this kind of *either-or* condition by providing an *else* clause. The syntax of the *if-else* statement is the following :

```
if (expression)
    statement 1;
else
    statement 2;
```

If the *expression* evaluates to *true i.e.*, a nonzero value, the *statement-1* is executed, otherwise, *statement-2* is executed. The *statement-1* and *statement-2* can be a single statement, or a compound statement, or a null statement.

note

Remember, in an *if-else* statement, only the code associated with *if* (i.e., *statement-1*) or the code associated with *else* (i.e., *statement-2*) executes, never both.

Following figure (Fig. 6.4) illustrates *if* and *if-else* constructs of Java :

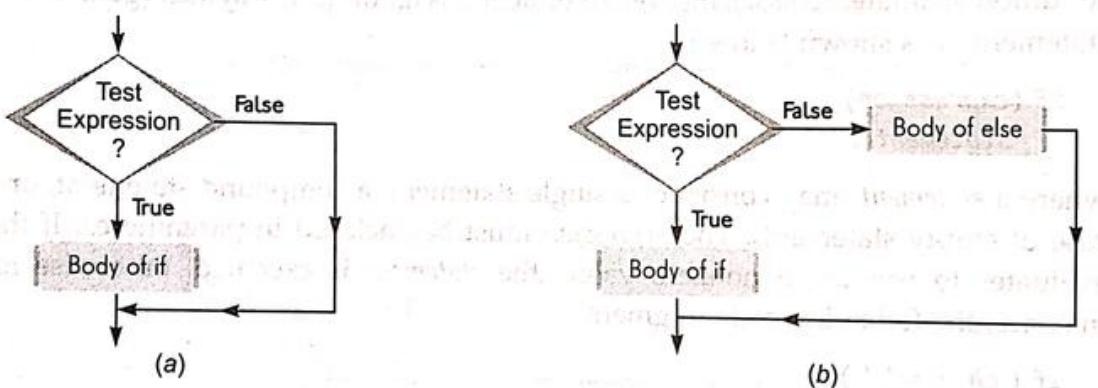


Figure 6.4 (a) The *if* statement's operation (b) The *if-else* statement's operation

Following example programs illustrate the syntax and working of *if* and *if-else* statements.

Program 6.1

Program to accept three integers and print the largest of the three. Make use of only *if* statement.

```
public class ex11_1 {
    public void Test(int x, int y, int z) {
        int max = 0 ;
        max = x ;
        if (y > max)
            max = y ;

        if (z > max)
            max = z ;
    }
}
```

```
System.out.println("Largest number is " + max ) ;
```

```

public static void main(String[ ] args) {
    ex11_1 obj1 = new ex11_1( ) ; // object created from class ex11_1
    obj1.Test (4, 5, 6) ; // member method Test( ) invoked through object
}
}

```

The output of above program will be as :

Largest number is 6

Program 6.2

Temperature-conversion program that gives the user the option of converting Fahrenheit to Celsius or Celsius to Fahrenheit and depending upon user's choice carries out the conversion. User can choose to execute desired function, by sending 1 for $^{\circ}\text{C}$ to $^{\circ}\text{F}$ conversion or any other no. for $^{\circ}\text{F}$ to $^{\circ}\text{C}$ conversion in a method namely Choice()

```

public class ex11_2 {
    public void FahToCelsius( float tempF) {
        float tempC = (tempF-32) / 1.8F;
        System.out.println("Equivalent temperature of " + tempF +
                           "F in Celsius is " + tempC);
    }
    public void CelToFahrenheit( float tempC) {
        float tempF = (tempC *1.8F) +32 ;
        System.out.println("Equivalent temperature of " + tempC +
                           "C in Fahrenheit is " + tempF);
    }
    public void Choice(int i) {
        ex11_2 obj2 = new ex11_2( );
        if (i==1) {
            obj2.CelToFahrenheit (37.0F) ; // converting 37 C into fahrenheit
        }
        else {
            obj2.FahToCelsius (98.4F) ; // converting 98.4 F into celcsius
        }
    }
    public static void main(String[ ] args) {
        ex11_2 obj1 = new ex11_2( );
        System.out.println("Converting a temperature from Celsius to Fahrenheit...") ;
        obj1.Choice(1) ;
        System.out.println("\nConverting a temperature from Fahrenheit to Celsius...") ;
        obj1.Choice(2) ;
    }
}

```

The output of above program will be as :

Converting a temperature from Celsius to Fahrenheit...

Equivalent temperature of 37.0C in Fahrenheit is 98.6

Converting a temperature from Fahrenheit to Celsius...

Equivalent temperature of 98.4F in Celsius is 36.88889

Notice that the *test condition* of if statement can be any *relational expression* or a *logical statement* (i.e., a statement that results into either *true* or *false*). The *test condition* of if must be enclosed in parenthesis. Now notice that in *program 6.1*, the executable part of if is not enclosed in curly braces ({}) whereas in *program 6.2*, the executable codes of if and else parts are enclosed in curly braces ({}). You must know the reason for this : the braces ({}) are not required if only **ONE statement follows if or else**, however, it is a good idea to put the executable code of if/else in curly braces {} .

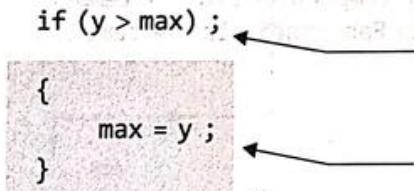
That means, the if statements of program 6.1 can also be written as :

```
if (y > max) {
    max = y ;
}
if (z > max) {
    max = z ;
}
```

Important Placement of semicolon is also important. In an if statement, DO NOT put semicolon in the line having test condition, such as

`if (y > max) ;`  *Do not do this.*

In such a case, the if will stop at this very semicolon and the statements following this line will not be considered part of if. That is, if you have written a code like :

`if (y > max) ; {
 max = y ;
}`  *if stops here*

These are not part of if. These statements will be executed even if the condition $y > max$ is false or true, i.e., these statements will be executed irrespective of the result of test condition $y > max$.

Therefore, make sure **not to put semicolon after the test condition**.

Caution If you put a semicolon after the test condition, the if statement ends there. The block or ! statements following are no more part of if in such cases.

Another important thing, that you must know is that the expression in if (i.e., in *if(expression)*) must be of *Boolean type*. If you specify any other type of expression, it will result into error.

For instance,

`int x = 10 ;
if (x) {
 :
}
if (x >= 10) {
 :
}`

 *will cause error as x is of int type not boolean*

 *is OK as $x \geq 10$ is a boolean expression*

Note The expression of if must be of boolean type.

6.3.1A Nested Ifs

A *nested if* is an if that has another if in its *if's body* or in its *else's body*. The *nested if* can have one of the following three forms :

1. if nested inside if-part

```
if (expression1)
{
    if (expression2)
        statement 1;
    [else
        statement-2];
    :
}
else
    body-of-else ;
```

2. if nested inside else-part

```
if (expression1)
    body-of-if ;
else
    { : if (expression2)
        statement-1;
    [ else
        statement-2];
    :
}
```

3. if nested inside both if-part and else-part

```
if (expression1)
{
    if (expression2)
        statement-1;
    [else
        statement-2];
    :
}
else
{
    if (expression3)
        statement-3;
    [else statement-4];
    :
}
```

See, optional parts are shown shaded

In an *if* statement, either there can be *if* statement(s) in its *body-of-if* or in its *body-of-else* or in both.

The part in [] means, it is optional. The inner *ifs* can themselves be *nested ifs*, but the inner *if* must terminate before an outer *if*. Following example programs illustrate the use of *nested ifs*.

Program 6.3

Program to create the equivalent of a four-function calculator. The program requires the user to enter two numbers and an operator as arguments. It then carries out the specified arithmetical operation : addition, subtraction, multiplication or division of the two numbers. Finally, it displays the result.

```
public class ex11_3 {
    public void Calculator( double a, double b, char oper) {
        double result = 0 ;
        if (oper == '+')
            result = a + b ;
        else if (oper == '-')
            result = a - b ;
        else if (oper == '*')
            result = a * b ;
        else if (oper == '/')
            result = a / b ;
        else
            System.out.println("Wrong operator !!");
        System.out.println("Calculated result is " + result);
    }
}
```

```

        public static void main(String[ ] args) {
            ex11_3 obj1 = new ex11_3( );           // object of ex11_3 class created
            System.out.println("\nExecuting + with 135.74 and 14.26");
            obj1.Calculator (135.74, 14.26, '+');
            System.out.println ("\nExecuting - with 135.74 and 14.26");
            obj1.Calculator (135.74, 14.26, '-');
            System.out.println ("\\nExecuting * with 35.74 and 14.26");
            obj1.Calculator (35.74, 14.26, '*');
            System.out.println ("\\nExecuting / with 635.544 and 24.16");
            obj1.Calculator(635.544, 24.16, '/');
        }
    }

Executing + with 135.74 and 14.26
Calculated result is 150.0

Executing - with 135.74 and 14.26
Calculated result is 121.48

Executing * with 35.74 and 14.26
Calculated result is 509.6524

Executing / with 635.544 and 24.16
Calculated result is 27.050662251655627

```

Program 6.4

Program to input a character and to print whether a given character is an alphabet, digit or any other character.

```

public class ex11_4 {
    public void Test(char ch) {
        if ( (ch >= 'A' && ch <= 'Z')||(ch >= 'a' && ch <= 'z') )
            System.out.println("You entered an alphabet.");
        else if (ch >= '0' && ch <= '9')
            System.out.println("You entered a digit.");
        else
            System.out.println("You entered a special character.");
    }

    public static void main(String [ ] args) {
        ex11_4 obj1 = new ex11_4( );
        System.out.println("\nPassing 'D' to Test( )");
        obj1.Test( 'D' );
        System.out.println("\nPassing 'h' to Test( )");
        obj1.Test( 'h' );
        System.out.println("\nPassing '7' to Test( )");
        obj1.Test( '7' );
        System.out.println("\nPassing '$' to Test( )");
        obj1.Test( '$' );
    }
}

```

The output produced is

Passing 'D' to Test()
You entered an alphabet.

Passing 'h' to Test()
You entered an alphabet.

Passing '7' to Test()
You entered a digit.

Passing '\$' to Test()
You entered a special character.

The **nested if-else** statement introduces a source of potential ambiguity referred to as **dangling-else** problem. This problem arises when in a *nested if* statement, number of **ifs** is more than the number of **else** clauses. The question then arises, with which **if** does the additional **else** clause property matchup. For instance,

```
if ( ch >= 'A' ) // outer if
    if ( ch <= 'Z' ) // inner if
        ++upcase ;
    else
        ++others ;
```

The indentation in the above code fragment indicates that programmer wants the **else** to be with the outer **if**. However, Java matches an **else** with the preceding unmatched **if**. In this case, the actual evaluation of the **if-else** statement will be as shown below :

```
if ( ch >= 'A' ) // enter if
    if ( ch <= 'Z' ) // inner if
        ++upcase ;
    else
        ++others ;
```

This else goes with the inner if.

That is, if the inner expression is *false* i.e., zero then the **else** clause gets excited. So, here is a tip for you.

TIP In nested if statement, a dangling else statement goes with the preceding unmatched if statement.

In an expression such as given below :

```
if (expr 1)
    if (expr 2)
        statement-1 ;
    else
        statement-2 ;
```

the last **else** statement goes with the *immediately preceding if* statement that does not already have an **else** statement. Here, **statement-2** is executed if **expr 2** evaluates to *false*. However, if you have a code as follows :

```
if (expr 1)
    if (expr 2)
        statement-1 ;
    else
        statement-2 ;
else
    statement-3 ;
```

the **inner else** goes with immediately preceding unmatched **if** which is the **inner if**. The **outer else** goes with immediately preceding unmatched **if** which is now **outer if** as the **inner if** is matched. Thus, **statement-3** gets executed if **expr1** is *false*.

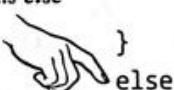
What if you want to override the default *dangling-else* matching? That is, if you have a code as shown below:

```
if (exp1)
    if (exp2)
        statement-1 ;
    else
        statement-2 ;
```

And you want the *else* to go with the other *if*. (By default, it will go with the inner *if*).

One method of over-riding the default *dangling-else* matching is to place the last occurring unmatched *if* in a compound statement, as it is shown below:

```
if (expr 1)      { // outer if
    if (expr2) // inner if
        statement-1 ;
    }
Now that inner if is
enclosed with { }, this else
goes with outer if.
    else
}
statement-2 ;
```



Now, the *else* clause matches up with the outer *if* as intended. So, another tip for you is:

TIP

Override the default dangling-else matching by using braces { }.

Program 6.5

Program to calculate commission for the salesmen. The commission is calculated according to following rates:

Sales	Commission Rate
30001 onwards	15 %
22001 – 30000	10 %
12001 – 22000	7 %
5001 – 12000	3 %
0 – 5000	0 %

The program accepts the sales made by the salesman and displays the calculated commission.

```
public class ex11_5 {
    public void calComm(float sales) {
        double comm = 0 ;
        if (sales > 5000)
        {
            if (sales > 12000)
            {
                if (sales > 22000)
                {
```

```

        if (sales > 30000)
            comm = sales * 0.15 ;
        else
            comm = sales * 0.10 ;
    }
    else
        comm = sales * 0.07 ;
}
else
    comm = sales * 0.03 ;
}
else
    comm = 0 ;
System.out.println("The commission is Rs. " + comm) ;
}

public static void main(String[ ] args)
{
    ex11_5 obj1 = new ex11_5( ) ;
    System.out.println("Calculating commission for 4678") ;
    obj1.calComm(4678f) ;
    System.out.println("\nCalculating commission for 40750") ;
    obj1.calComm(40750f) ;
    System.out.println("\nCalculating commission for 14800") ;
    obj1.calComm(14800f) ;
    System.out.println("\nCalculating commission for 24600") ;
    obj1.calComm(24600f) ;
}
}

```

calculating commission for 4678

The commission is Rs. 0.0

calculating commission for 40750

The commission is Rs. 6112.5

calculating commission for 14800

The commission is Rs. 1036.0

calculating commission for 24600

The commission is Rs. 2460.0

6.3.1B The if-else-if Ladder

A common programming construct in Java is the *if-else-if ladder*, which is often also called the *if-else-if staircase* because of its appearance.

It takes the following general form :

```

if (expression 1) statement 1 ;
else

```

```

if (expression 2) statement 2 ;
else
    if (expression 3) statement 3 ;
    :
    else statement 4 ;

```

The expressions are evaluated from the top downward. As soon as an *expression* evaluates to *true*, the *statement* associated with it is executed and the rest of the ladder is bypassed. If none of the *expressions* are *true*, the final *else* gets executed. If the final *else* is missing, no action takes place if all other conditions are *false*.

Although it is technically correct to have indentation in the *if-else-if ladder* as shown above, however, it generally leads to overly deep indentation. For this reason, the *if-else-if ladder* is generally indented like this :

```

if (expression 1)
    statement 1 ;
else if (expression 2)
    statement 2 ;
else if (expression 3)
    statement 3 ;
    :
    :
else
    statement n ;

```

Program 6.6

Program to print whether a given character is an uppercase or a lowercase character or a digit or any other character.

```

public class ex11_6 {
    public void Test(char ch) {
        if (ch >= 48 && ch <= 57)
            System.out.println("You entered a digit.");
        else if (ch >= 65 && ch <= 90)
            System.out.println("You entered an UPPERCASE character.");
        else if (ch >= 97 && ch <= 122)
            System.out.println("You entered a lowercase character.");
        else
            System.out.println("You entered a Special character.");
    }
    public static void main(String[ ] args) {
        ex11_6 obj1 = new ex11_6();
        System.out.println("\nPassing 'D' to Test()");
        obj1.Test('D');
        System.out.println("\nPassing 'h' to Test()");
    }
}

```

```

        obj1.Test( 'h' );
        System.out.println("\nPassing \'7\' to Test( )");
        obj1.Test( '7' );
        System.out.println("\nPassing '\$\' to Test( )");
        obj1.Test( '$' );
    }
} // end of class ex11_6

```

Passing 'D' to Test()
 You entered an UPPERCASE character.
 Passing 'h' to Test()
 You entered a lowercase character.
 Passing '7' to Test()
 You entered a digit.
 Passing '\$' to Test()
 You entered a Special character.

Program 6.7

Program to calculate and print roots of a quadratic equation $ax^2 + bx + c = 0$ ($a \neq 0$). The coefficients of quadratic equations a, b, c – are received as parameters.

```

public class ex11_7 {
    public void Test(double a, double b, double c) {
        double d, root1, root2 ;
        if (a==0)
            System.out.println("Value of a should not be zero. Aborting!!!");
        else {
            d = b*b - 4*a*c ;
            if (d>0) {
                root1 = (- b + Math.sqrt(d))/(2*a) ;
                root2 = (- b - Math.sqrt(d))/(2*a) ;
                System.out.println("Roots are real and unequal.");
                System.out.println("Root1 : "+root1+" \tRoot2 : "+root2);
            }
            else if (d==0) {
                root1 = - b / (2*a) ;
                root2 = root1 ;
                System.out.println("Roots are real and equal.");
                System.out.println("Root1 : "+root1+" \tRoot2 : "+root2);
            }
            else
                System.out.println("Roots are complex and imaginary.");
        }
    }
    public static void main(String[ ] args) {
        ex11_7 obj1 = new ex11_7( );
    }
}

```

```

        System.out.println("\nPassing a, b, c as 1 , 3, 2") ;
        obj1.Test(1.0, 3.0, 2.0) ;
        System.out.println("\nPassing a, b, c as 2 , 4, 2") ;
        obj1.Test(2.0, 4.0, 2.0) ;
        System.out.println("\nPassing a, b, c as 6 , 4, 3") ;
        obj1.Test(6.0, 4.0, 3.0) ;
    }
}

```

Passing a,b,c as 1 , 3, 2
 Roots are real and unequal.
 Root1 : -1.0 Root2 : -2.0
 Passing a,b,c as 2 , 4, 2
 Roots are real and equal.
 Root1 : -1.0 Root2 : -1.0
 Passing a,b,c as 6 , 4, 3
 Roots are complex and imaginary.

Caution!! a Careless Mistake May Go Unnoticed

While comparing for equality, make sure to use two equal (=) signs i.e., == because one equal (=) sign means assignment.

`if (p = 4)`

is almost ALWAYS true. The condition

`p = 4`

is actually an assignment statement (it assigns 4 to p). If the assignment value is 0, as in `if (p = 0)`, the variable p will be assigned value 0.

On the contrary, `if (p = 4)` is *true* ONLY if the value stored in p is the number 4. Thus, be careful while typing conditions for equality.

Note ONE equal (=) sign is used to assign a value, but Two equal (=) signs (i.e., ==) are used to check to see if values are equal to one another.

6.3.1C The ?: Alternative to if

Java has an operator that can be used as an alternative to if statement. You are familiar with this operator, the conditional operator ?: This operator can be used to replace if-else statements of the general form :

```

if (expression1)
  expression2 ;
else
  expression3 ;

```

The above form of if can be alternatively written using ?: as follows :

`expression1 ? expression2 : expression3 ;`

It works in the same way as the above given form of if does i.e., expression1 is evaluated, if it is true, expression2 gets executed (i.e. its value becomes the value of entire expression) otherwise expression3 gets executed (i.e., its value now becomes the value of the entire expression). For instance, the following if statement

```
int c ;
if ( a > b )
    c = a ;
else
    c = b ;
```

can be alternatively written as

```
int c = a > b ? a : b ;
```

This condition, if results to true, then value between ? and : is considered otherwise value following : is considered.

See how simple and compact your code has become.

Comparing if and ?:

1. Compared to if-else sequence, ?: offers more concise, clean and compact code, but it is less obvious as compared to if.
2. Another difference is that the conditional operator ?: produces an expression, and hence a single value can be assigned or incorporated into a larger expression, whereas, if is more flexible. The if statement can have multiple statements, multiple assignments and expressions (in form of a compound statement) in its body.
3. When ?: operator is used in its nested form, it becomes complex and difficult to understand. This form of ?: is generally used to conceal the purpose of code.

6.3.2 The switch Statement

Java provides a multiple-branch selection statement known as switch. This selection statement successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.

The syntax of switch statement is as follows :

```
switch (expression)
{
    case constant1 : statement sequence 1 ;
                      break ;
    case constant2 : statement sequence 2 ;
                      break ;
    case constant3: statement sequence 3 ;
                      break ;
    :
    case constant n-1 : statement sequence n-1 ;
                      break ;
    [default :           statement sequence n] ;
}
```

The expression is evaluated and its values are matched against the values of the constants specified in the case statements. When a match is found, the statement sequence associated with

that case is executed until the **break** statement or the end of **switch** statement is reached. The **default** statement gets executed when no match is found. If the control flows to the next case below the matching case, in the absence of **break**, this is called **fall through**. The **default** statement is optional and, if it is missing, no action takes place if all matches fail.

If you do not give **break** after each case, then Java will start executing the statements from matching case and keep on executing statements for following cases as well until either a **break** statement is found or switch statement's end is encountered. If the control flows to the next case below the matching case, in the absence of **break**, this is called **fall through**.

A **case** statement cannot exist by itself, outside of a **switch**. The **break** statement, used under **switch**, is one of Java jump statements. (You'll learn more about it later in the chapter.) When a **break** statement is encountered in a **switch** statement, program execution jumps to the line of code following the **switch** statement i.e., outside the body of **switch** statement.

Note The data type of expression in a switch must be **byte**, **char**, **short** or **int**.

The fall of control to the following cases of matching case, is called **FALL-THROUGH**.

Program 6.8

Program to input number of week's day (1-7) and translate to its equivalent name of the day of the week. (e.g., 1 to Sunday, 2 to Monday, . . . , 7 to Saturday).

```
public class ex11_8 {
    public void WeekDay( int dow ) {
        String ans ;
        switch (dow) {
            case 1 : ans= "Sunday" ;
            break ;
            case 2 : ans= "Monday" ;
            break ;
            case 3 : ans= "Tuesday" ;
            break ;
            case 4 : ans= "Wednesday" ;
            break ;
            case 5 : ans= "Thursday" ;
            break ;
            case 6 : ans= "Friday" ;
            break ;
            case 7 : ans= "Saturday" ;
            break ;
            default : ans = "Wrong day number" ;
        }
        System.out.println(ans) ;
    }
    public static void main(String[ ] args) {
        ex11_8 obj1 = new ex11_8( ) ;
    }
}
```

```

        System.out.print("\nWeekday for day no 6 is :");
        obj1.WeekDay(6);
        System.out.print("\nWeekday for day no 4 is :");
        obj1.WeekDay(4);
        System.out.print("\nWeekday for day no 11 is :");
        obj1.WeekDay(11);
    }
}

```

weekday for day no 6 is : Friday

weekday for day no 4 is : Wednesday

weekday for day no 11 is : Wrong day number

Let us consider another example program illustrating use of **switch statement**.

Program 6.9

Program to calculate area of a circle, a rectangle or a triangle depending upon user's choice passed as 1, 2 or 3 for area of circle, rectangle and triangle respectively in a method namely choice()

```

public class ex11_9 {
    public void CircleArea( double r) {
        double area = 3.14 * r * r;
        System.out.println(area);
    }
    public void RectangleArea( double l, double b) {
        double area = l * b ;
        System.out.println(area);
    }
    public void TriangleArea( double a, double b, double c) {
        double s, area ;
        s = (a + b + c)/2 ;
        area = Math.sqrt( s *(s - a) * (s - b) * (s - c)) ;
        System.out.println(area);
    }
    public static void main(String[ ] args) {
        ex11_9 obj1 = new ex11_9( );
        // passed argument args[0] stores the choice
        char ch = args[0].charAt(0); // charAt(0) gives the very first character of a string
        switch(ch)
        {
            case '1' : System.out.print("Area of Circle with radius 3.5 is ");
                         obj1.CircleArea(3.5);
                         break;
            case '2' : System.out.print("\nArea of Reactangle with length = 3.5,
                                         breadth = 4.3 is ");
                         obj1.RectangleArea(3.5, 4.3);
                         break;
        }
    }
}

```

```
        case '3' : System.out.print("\nArea of Triangle with sides 2, 3, 4 is ");
                     obj1.TriangleArea(2.0, 3.0, 4.0);
                     break;
        default : System.out.println("Wrong choice! Only 1,2,3 are allowed as choice.");
    }
}
```

Area of Circle with radius 3.5 is 38.465

Area of Rectangle with length = 3.5, breadth = 4.3 is 15.04999999999999

Area of Triangle with sides 2, 3, 4 is 2.9047375096555625

Two more things that you should know of switch are that ***default case of switch need not be the last one***. It can be anywhere in the switch. Another thing is that ***there must not be two more identical cases***. It would lead to error. Consider the following code fragments.

```
switch (val) {  
    case 5 :  
        ;  
    case 7 :  
        ;  
    case 9 :  
        ;  
    case 5 :  
        ;  
}
```

Two identical case expressions are not allowed.

6.3.2A The switch Vs. if-else

The **switch** and **if-else** both are selection statements and they both let you select an alternative out of given many alternatives by testing an expression. However, there are some differences in their operations. These are given below :

1. The **switch** statement differs from the **if** statement in that **switch** can only test for *equality* whereas **if** can evaluate a relational or logical expression i.e., multiple conditions.
 2. The **switch** statement selects its branches by testing the value of same variable (against a set of constants) whereas the **if-else** construction lets you use a series of expressions that may involve unrelated variables and complex expressions.
 3. The **if-else** is more versatile of the two statements. For instance, **if-else** can handle ranges whereas **switch** cannot. Each **switch case** label must be a single value.
 4. The **if-else** statement can handle *floating-point tests* also apart from handling *integer* and *character* tests whereas a **switch** cannot handle floating-point tests. The *case labels* of **switch** must be an integer *byte, short, int* or a *char*.

5. The switch case label value must be a constant. So, if two or more variables are to be compared, use if-else.

6. The switch statement is more efficient choice in terms of code used in a situation that supports the nature of switch operation (testing a value against a set of constants).

6.3.2B The Nested-Switch

Like if statements, a switch statement can also be nested. There can be a switch as part of the statement sequence of another switch. For instance, the following code fragment is perfectly all right in Java.

```

switch (a) {
    case 1 : switch (b) {
        case 0 : System.out.println ("Divide by zero-Error ! ! ");
        break ;
        case 1 : res = a/b ;
    }
    break ;
    case 2 :
    :
}

```

*Switch nested inside case 1 of
outer switch.*

6.3.2C Some Important Things to Know about switch

Following important things you must know about the switch statement :

1. A switch statement can not work for non-equality comparisons.
2. The case labels of switch statements must be literals or constants.
3. No two case labels in the same switch can have identical values. But, in case of nested switch statements the case constants of the inner and outer switch can contain common values.
4. Switch statement works with integral types (byte, short, int, long) and char only.
5. The switch statement is more efficient than if in a situation that supports the nature of switch operation.

TIP A switch statement is more efficient than nested if-else statement.

For instance, a statement that tests a value against a set of constants like this :

```

if ( wish == 'a' ) {
    :
}
else if ( wish == 'b' ) {
    :
}
else if ( wish == 'c' ) {
    :
}
else {
    :
}

```

is better written as a **switch** statement as shown below :

```
switch (wish) {  
    case 'a' : ;  
    break ;  
    case 'b' : ;  
    break ;  
    case 'c' : ;  
    break ;  
    default : ;  
    break ;  
}
```

A **switch** statement only evaluates the expression once, while the **if** statement will evaluate it repeatedly until it finds a match. Thus, a **switch** is an efficient choice in such a situation. There is another thing that I would like you to keep in mind that helps in better programming. And for that read the tip below :



Always put a break statement after the last case statement in a switch.

Although it is not necessary to put a **break** after the last statement in a switch, since control will leave the statement anyway, yet it should be done to avoid forgetting the **break** when you add another **case** statement at the end of the switch.

Following example programs 6.10 and 6.11 illustrate the working in case of absence of *break* statements in a *switch* and in presence of *break* statements in a *switch*.

 Program 6.10

Program to illustrate the working of switch in the absence of break statement.

```

public static void main(String [ ] args) {
    ex11_10 obj1 = new ex11_10( );
    System.out.println("Value of i being passed as 1");
    obj1.switchTest( 1 );
    System.out.println("\nValue of i being passed as 3");
    obj1.switchTest( 3 );
    System.out.println("\nValue of i being passed as 0");
    obj1.switchTest( 0 );
}
}

```

The above program produces the following output :

value of i being passed as 1

ua incremented.

ub incremented.

uc incremented.

fail incremented.

ua = 1 ub = 1

uc = 1 fail = 1

value of i being passed as 3

ub incremented.

uc incremented.

fail incremented.

ua = 0 ub = 1

uc = 1 fail = 1

value of i being passed as 0

uc incremented.

fail incremented.

ua = 0 ub = 0

uc = 0 fail = 1

This is because when case 1 is matched, all the statements till the end of *switch* ($++ua$, $++ub$, $++uc$, $++fail$) are executed as there is no *break* statement. Remember, when a match is found, then the statements corresponding to that match are executed along with all the statements following it until a *break* statement or end of *switch* is encountered. Therefore, the above program works in the following fashion :

When i is passed as	Match	The statements executed are	Values after this step			
			ua	ub	uc	fail
1.	case 1	$++ ua$, $++ub$, $++uc$, $++fail$	1	1	1	1
2.	case 3	$++ub$, $++uc$, $++fail$	0	1	1	1
3.	default	$++fail$	0	0	0	1

Now if you insert *break* statements in each *case* of *switch*, the above program shall behave differently, following program illustrates this.

Program 6.11

Program to illustrate the working of switch in the presence of break statements.

```
public class ex11_11 {
    public void switchTest( int i ) {
        int ua = 0, ub = 0, uc = 0, fail = 0 ;
        switch(i++) {
            case 1:
                case 2: ++ua ;
                    System.out.println(" ua incremented. ") ;
                    break ;
            case 3:
            case 4: ++ub ;
                    System.out.println(" ub incremented. ") ;
                    break ;
            case 5: ++uc ;
                    System.out.println(" uc incremented. ") ;
                    break ;
            default: ++fail ;
                    System.out.println(" fail incremented. ") ;
        }
        System.out.println("ua = " + ua + "\tub = " + ub + "\nuc = " + uc + "\tfail = " + fail) ;
    }
    public static void main(String [ ] args) {
        ex11_11 obj1 = new ex11_11( );
        System.out.println("Value of i being passed as 1") ;
        obj1.switchTest( 1 ) ;
        System.out.println("\nValue of i being passed as 3") ;
        obj1.switchTest( 3 ) ;
        System.out.println("\nValue of i being passed as 0") ;
        obj1.switchTest( 0 ) ;
    }
}
```

Value of i being passed as 1

ua incremented.
ua = 1 ub = 0
uc = 0 fail = 0

Value of i being passed as 3

ub incremented.
ua = 0 ub = 1
uc = 0 fail = 0

value of i being passed as 0

uc incremented.
ua = 0 ub = 0
uc = 0 fail = 1

This is because when a match is found, only the statements corresponding to it are executed as each case is followed by a *break* statement.

Getting Input via Keyboard

Until now, you have seen the programs where the values being processed are provided via parameters or by assigning values to variables. If you want to obtain the input via keyboard then you may follow the steps given below. We are giving only the steps without explanations as a later chapter (Chapter 10) talks about these concepts in details. For now, you just need to learn the steps given below :

1. Import Java's IO library by giving following statement at the top of your program:

```
import java.io.*;
```

2. Create an input stream by giving following statement :

```
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
```

The above statement creates an input stream namely `stdin`, which you'll be using for obtaining input.

3. You can then read a line of input like this :

```
String line = stdin.readLine();
```

See the same input stream (`stdin` that we created) is now used to get input.

4. To read an integer value, use statement as given here:

```
int number = Integer.parseInt(stdin.readLine());
```

5. To read a float value, use statement as given here:

```
float val = Float.parseFloat(stdin.readLine());
```

6. To read a double value, use statement as given here:

```
double number = Double.parseDouble(stdin.readLine());
```

7. To read a character value, use statement as given here:

```
char ch = (char)(stdin.read());
```

8. And while reading values through key board either include

```
throws IOException
```

in the first line (i.e., header) of the function. Or write input statements inside a `try` block. Each `try` block must be followed by a `catch` block. If you are using `try` and `catch` blocks then you can skip the keywords `throws IOException`. (Some Solved problems (23 - 26) have been given at the end of the chapter that use `try` and `catch` blocks.)

Now consider the following example code :

```
import java.io.*;
class InputExample {
    static BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
    public static void main(String [] args) throws IOException {
        BufferedReader stdin = new BufferedReader( new InputStreamReader(System.in));
        System.out.print("What is your name? ");
        String name = stdin.readLine();
        System.out.print("And what is your age? ");
        int age = Integer.parseInt(stdin.readLine());
        System.out.print(name + ", you are " + age + " years old.");
    }
}
```

L et Us Revise

- ❖ Statements are the instructions given to the computer to perform any kind of action.
- ❖ The flow of control in a program can be in three ways : sequentially (the sequence construct), selectively (the selection construct), and iteratively (the iteration construct).
- ❖ The sequence construct means statements get executed sequentially.
- ❖ The selection construct means the execution of statement(s) depending upon a condition-test.
- ❖ The iteration construct means repetition of a set-of-statements depending upon a condition-test.
- ❖ Java provides two types of selection statements : if and switch.
- ❖ The if-else statement tests an expression and depending upon its truth value one of the two sets-of-action is executed.
- ❖ The if-else statement can be nested also i.e., an if statement can have another if statement.
- ❖ In a nested if-else statement, an else goes with immediately preceding unmatched if.
- ❖ The conditional operator ?: can be used as an alternative to if-else.
- ❖ Java provides one more selection statement known as switch that tests a value against a set of integer constants (that includes characters also).
- ❖ A switch statement can be nested also.
- ❖ An if-else statement is more flexible and versatile compared to switch but switch is more efficient in a situation when the same variable is compared against a set of values for equality.

6.4 Iteration Statements

The iteration statements allow a set of instructions to be performed repeatedly until a certain condition is fulfilled. The iteration statements are also called *loops* or *looping statements*. Java provides *three* kinds of loops : for loop, while loop, and do-while loop.

All *three* loop constructs of Java repeat a set of statements as long as a specified condition remains true. This specified condition is generally referred to as a *loop control*. For all three loop statements, a *true condition* is the one that returns boolean *true* value and the *false condition* is the one that returns boolean *false* value.

Let us begin with elements of a loop.

6.5 Elements that Control a Loop (Parts of a Loop)

Every loop has its elements that control and govern its execution. Generally, a loop has *four* elements that have different purposes. These elements are as given below :

1. Initialization Expression(s). Before entering in a loop, its control variable(s) must be initialized. The initialization of the control variable(s) takes place under *initialization expression(s)*. The initialization expression(s) give(s) the loop variable(s) their first value(s). The initialization expression(s) is *executed only once*, in the beginning of the loop.

2. Test Expression. The *test expression* is an expression whose truth value decides whether the *loop-body* will be executed or not. If the *test expression* evaluates to *true* i.e., 1, the *loop-body* gets executed, otherwise the loop is terminated.

In an *entry-controlled loop*, the *test-expression* is evaluated before entering into a loop whereas in a *exit-controlled loop*, the *test-expression* is evaluated before exiting from the loop. In Java, the **for** loop and **while** loop are *entry-controlled* loops and **do-while** loop is *exit-controlled* loop.

3. Update Expression(s). The update expression(s) change the value(s) of loop variable(s). The update expression(s) is executed ; at the end of the loop after the loop-body is executed.

4. The Body-of-the-Loop. The statements that are executed repeatedly (as long as the test-expression is nonzero) form the body of the loop. In an *entry-controlled* loop, first the *test-expression* is evaluated and if it is nonzero, the *body-of-the-loop* is executed ; if the *test-expression* evaluates to be zero, the loop is terminated. In an *exit-controlled* loop, the *body-of-the-loop* is executed first and then the *test-expression* is evaluated. If it evaluates to be zero, the loop is terminated, otherwise repeated.

6.6 The for Loop – Fixed Number of Iterations

The **for** loop is the easiest to understand of the Java loops. All its loop-control elements are gathered in one place (on the top of the loop), while in the other loop construction of Java, they (top-control elements) are scattered about the program. The general-form (syntax) of the **for** loop statement is

```
for (initialization expression(s) ; test-expression ; update expression(s))
    body-of-the-loop ;
```

The following example program illustrates the use of **for** statement.

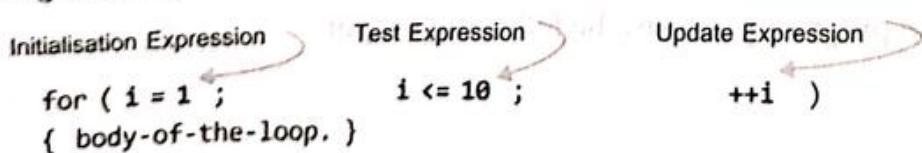
Program 6.12

Program using **for** loop to print numbers from 1 to 10.

```
public class ex11_12 {
    public void printNos() {
        int i = 0;
        for( i = 1 ; i <= 10 ; ++i ) {
            System.out.print( i + " " );
        }
    }
    public static void main(String[ ] args) {
        ex11_12 obj1 = new ex11_12();
        obj1.printNos();
    }
}
```

The output produced is
1 2 3 4 5 6 7 8 9 10

The following lines explain the working of the above given **for** loop.



1. Firstly, *initialization expression* is executed i.e., $i=1$ which gives the first value 1 to variable i .
2. Then, the *test expression* is evaluated i.e., $i \leq 10$ which results into true i.e., 1.

3. Since, the *test-expression* is true, the *body-of-the-loop* i.e., `System.out.print (i + " ")` is executed which prints the current value of *i* on the same line.
4. After executing the loop-body, the *update expression* i.e., `++i` is executed which increments the value of *i*. (After first execution of the loop, the value of *i* becomes 2 after the execution of `++i`, since initially *i* was 1).
5. After the *update expression* is executed, the *test-expression* is again evaluated. If it is *true* the sequence is repeated from step no 3, otherwise the loop terminates.

Following figure (Fig 6.5) outlines the working of a **for** loop.

Now that you are familiar with the working of a **for** loop. Let us take another example where there are multiple statements in the loop-body.

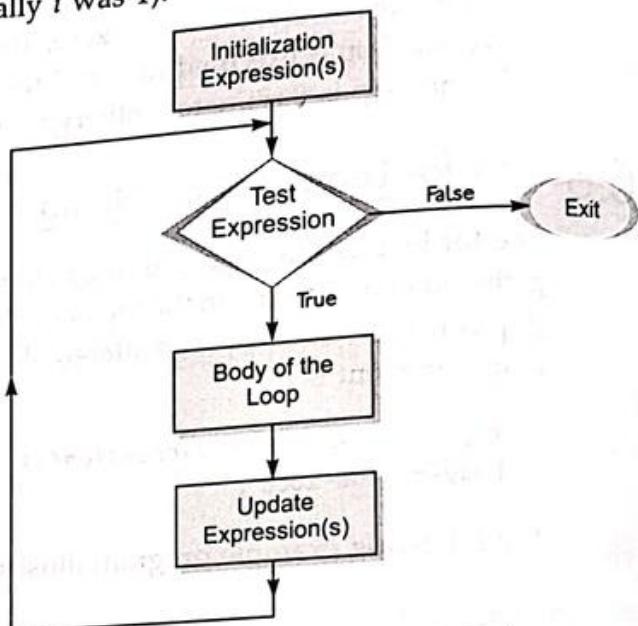


Figure 6.5 The execution of a for loop.

Program 6.13

Program to print first *n* natural numbers and their sum.

```

public class ex11_13 {
    public void Test(int n) {
        int i, sum;
        for( i = 1, sum = 0 ; i <= n ; ++i) {
            System.out.println(i);
            sum = sum + i;
        }
        System.out.println("The sum of first " + n + " natural numbers is " + sum);
    }
    public static void main (String[ ] args) {
        ex11_13 obj1 = new ex11_13( );
        obj1.Test(7);
    }
}
  
```

The above program produces the following output :

1
2
3
4
5
6
7

The sum of first 7 natural numbers is 28

In the above program, the *initialization expressions contain two expressions : i=1 and sum = 0 separated by comma. The initialization part may contain as many expressions but these should be separated by commas. The initialization part must be followed by a semicolon ; Both the variables i and sum get their first values 1 and 0 respectively. Then, the test expression i <= n is evaluated which on resulting true, initiates the execution of the loop-body (that contains two statements enclosed in a compound statement). After the execution of the loop-body, the update expression ++ i is executed. Then again the test-expression i <= n is evaluated depending upon whose truth value the loop is separated or terminated.*

Program 6.14

Program to tabulate the function : $f(x) = (x^2 + 1.5x + 5) / (x - 3)$

for x = - 10 to 10, x should take values - 10, - 8, - 6, , 6, 8, 10.

```
import java.io.*;
class ex11_14 {
    public void fnTest( int x ) {
        double fx ;
        for( x=-10 ; x <= 10 ; x = x + 2 ) {
            fx = (x*x + 1.5*x +5)/(x-3) ;
            System.out.println("For x = "+x+", f(x) is " + fx + "\n") ;
        }
    }
}
```

```
For x = -10, f(x) is - 6.923076923076923
For x = - 8, f(x) is - 5.181818181818182
For x = - 6, f(x) is - 3.5555555555555554
For x = - 4, f(x) is - 2.142857142857143
For x = - 2, f(x) is - 1.2
For x = 0, f(x) is - 1.6666666666666667
For x = 2, f(x) is - 12.0
For x = 4, f(x) is 27.0
For x = 6, f(x) is 16.666666666666668
For x = 8, f(x) is 16.2
For x = 10, f(x) is 17.142857142857142
```

TIP

Use for loop when you have to repeat a block of statements specific number of times.

6.6.1 The for Loop Variations

Our previous discussion about **for** loop used its most common form. However, Java offers several variations that increase the flexibility and applicability of **for** loop. The different variations of **for** loop are discussed below :

1. Multiple initialization and update Expressions

A **for** loop may contain multiple initialization and/or multiple update expressions. These multiple expressions must be separated by commas. We have seen an example of multiple initialization expressions in program 6.13. The **for** loop of the program 6.13 can be alternatively written as follows :

```
for (i = 1, sum = 0 ; i <= n ; sum += i , ++i )
    System.out.println( i );
```

The above code fragment contains two initialization expressions **i = 1** and **sum = 0** and two update expressions **sum += i** and **++i**. These multiple expressions are executed in sequence i.e., during initialization firstly **i=1** takes place followed by **sum = 0**. Similarly, during updation, firstly **sum += i** takes place followed by **++i**.

TIP The comma operator can serve in a **for** loop when you need more than one index.

2. Optional Expressions

In a **for** loop, *initialization expression*, *test expression* and *update expression* are optional i.e., you can skip any or all of these expressions. Say, for example, you already have initialized the loop variables and you want to scrap off the initialization expression then you can write **for** loop as follows :

```
for ( ; test-expression ; update-expression(s))
    Loop-body
```

See, even if you skip *initialization-expression*, but the *semicolon (;)* following it must be present. Following code fragment also skips the *initialization-expression* of a **for** loop.

```
int i = 1, sum = 0 ;
for ( ; i <= 20 ; sum += i, ++i )
    System.out.println( i );
;
```

Similarly, under certain circumstances the *update expression(s)* and the *test expression* can be omitted. For example, the following loop will run until the **j** becomes 242 :

```
for (j = 0 ; j != 242 ; )
    j += 11 ;
```

If the variable **j** has already been initialised before, then the above loop may be written as :

```
for ( ; j != 242 ; )
    j += 11 ;
```

TIP

The *loop-control expressions* in a **for** loop statement are optional, but semi-colons must be written.

3. Infinite Loop

Although any loop statement can be used to create an infinite (endless loop) yet **for** is traditionally used for this purpose. An infinite **for** loop can be created by omitting the *test-expression* as shown below :-

```
for ( j = 25 ; ; -- i )
    System.out.println ("An infinite for loop") ;
```

Similarly, the following **for** loop is also an infinite loop.

```
for ( ; ; )
    System.out.println ("Endless for loop") ;
```

4. Empty Loop

If a loop does not contain any statement in its loop-body, it is said to be an empty loop. In such cases, a Java loop contains an empty statement *i.e.*, a null statement. Following **for** loop is an empty loop :

```
for ( j = 20 ; (j >= 0) ; -- i )
```

See, loop-body contains a null statement

See, the body of the above **for** loop contains just (a null statement), it is an empty loop. An empty **for** loop has its applications in time delay loop where you need to increment or decrement some variable without doing anything else just for introducing some delay.

Time delay loops are often used in programs. The following code shows how to create one by using **for** :

```
for ( t = 0 ; t < 300 ; t++ )
```

5. Declaration of variables inside loops and if

A variable declared within an **if** or **for** / **while** / **do-while** statement cannot be accessed after the statement is over. The reason behind this is that the variable is declared within the block of statement, so, its scope becomes the body of the statement (**if** / **for** / **while** / **do-while**). Therefore, it cannot be accessed outside the statement body.

Following code fragment explains this concept :

```
if (ch == 'a') {
    int ans = 1 ;
    :
}
over
else {
    int ans = 2 ;
}
System.out.print (ans) ; //invalid !! scope of ans is over
```

for (int ans = 1 ; ans < 100 ; ans = ans + 20) {
 :
}

System.out.print (ans) ; //invalid !! scope of ans is over

In the above fragments, the statement **System.out.print(ans)** ; is invalid as the scope of **ans** *i.e.*, the block where it has been declared is over. A variable is not accessible outside its scope, thus, the mentioned statement is invalid.

6.7 The while Loop

The second loop available in Java is the **while** loop. The **while** loop is an *entry-controlled* loop. The syntax of a **while** loop is

```
while (expression)
    Loop-body
```

where the *loop-body* may contain a single statement, a compound statement or an empty statement. The loop iterates *while* the expression evaluates to *true*. When the expression becomes *false*, the program control passes to the line after the loop-body code.

In a **while** loop, a loop control variable should be *initialised* before the loop begins as an *uninitialised* variable can be used in an expression. *The loop variable should be updated inside the body-of-the-while*. Following example program illustrates the working of a **while** loop.

Program 6.15

Program to calculate the factorial of an integer.

```
public class ex11_15 {
    public void factorial(int num) {
        long i = 0, fact = 1 ;
        i = num ;
        while(num != 0) {
            fact = fact * num ;
            --num ;
        }
        System.out.println("The factorial of " + i + " is " + fact) ;
    }
    public static void main(String[ ] args) {
        ex11_15 obj1 = new ex11_15( ) ;
        obj1.factorial(5) ;
    }
}
```

The output produced is

The factorial of 5 is 120

The above program inputs an integer *num*. Then as long as *num* is nonzero (according to *while* (*num*)) the loop-body interates *i.e.*, *fact* is multiplied with *num* and the result is stored back in *fact*, followed by the decrement of *num*. Again the *test-expression* (*num*) is evaluated : If it is *true*, the loop is repeated otherwise terminated.

Program 6.16

Program to calculate and print the sums of even and odd integers of the first *n* natural numbers.

```
public class ex11_16 {
    public void SumOfNaturalNos(int n) {
        int sum_even = 0, sum_odd = 0, ctr = 1 ;
        while(ctr <= n) {
```

```

        if ( ctr % 2 == 0 )
            sum_even += ctr ;
        else
            sum_odd += ctr ;
        ctr++ ;
    }
System.out.println("The sum of even integers is "+ sum_even) ;
System.out.println("The sum of odd integers is "+ sum_odd) ;
}
public static void main(String[ ] args) {
    ex11_16 obj1 = new ex11_16( ) ;
    obj1.SumOfNaturalNos(25) ;
}
}

```

The output produced is

The sum of even integers is 156
The sum of odd integers is 169

6.7.1 Variations in a while Loop

A **while** loop can also have several variations. It can be an *empty loop* or an *infinite loop*. An *empty loop* does not contain any statement in its body other than the null statement i.e., just a semicolon (;). Following is an example of *empty loop* :

```

:
Long wait = 0 ;
while ( ++wait < 10000)           //null statement
;
:
```

The above given is a *time delay loop*. A *time delay loop* is useful for pausing the program for some time. For instance, if an important message is flashed on the screen and before you can read it, it goes off. Here you can introduce a *time delay loop* so that you can have sufficient time to read the message.

A **while** loop can be *infinite* if you forget to write the *update expression* inside its body. For instance, the following is an example of an *infinite while loop* :

```

j = 0 ;
while ( j <= n )
    System.out.println ( j * j ) ;      //single statement in the loop
j++ ;
:
```

The above loop is an *infinite loop* as (by default) a single statement is taken into a loop's body. Therefore, the increment statement `j++` is not included in the loop's body. The value of `j` remains the same and loop can never terminate. To avoid such a situation use a compound statement as shown below :

```

j = 0 ;
while ( j <= n )  {
    System.out.print ( j * j ) ;
    j++ ;
}

```

Now the above loop is a finite loop. It will terminate as soon as the value of `j` exceeds `n`.

6.8 The do-while Loop

Unlike the **for** and **while** loops, the **do-while** is an *exit-controlled* loop i.e., it evaluates its *test-expression* at the bottom of the loop after executing its loop-body statements. This means that a **do-while** loop always executes at least once.

In the other two loops **for** and **while**, the *test-expression* is evaluated at the beginning of the loop i.e., before executing the loop-body. If the *test-expression* evaluates to *false* for the first time itself, the loop is never executed. But in some situations, it is wanted that the loop-body is executed at least once, no matter what the initial state of the *test-expression* is. In such cases, the **do-while** loop is the obvious choice.

The syntax of the **do-while** loop is :

```
do { statement ;
} while (test-expression) ;
```

The braces {} are not necessary when the loop-body contains a single statement. The following **do-while** loop prints all upper-case letters :

```
char ch = 'A' ;
do {
    System.out.print( ch ) ;
    ch ++ ;
} while ( ch <= 'Z' ) ;
```

The above code characters from 'A' onwards until the condition $ch \leq 'Z'$ becomes false.

The most common use of the **do-while** loop is in menu selection routine, where the menu is flashed at least once. Then depending upon the user's response, it is either repeated or terminated. The following example program is a menu selection program.

Program 6.17

Program to display a menu regarding rectangle operations and perform according to user's response.

```
public class Ex17 {
    public void Area(float l, float b) {
        double area = l * b ;
        System.out.println("Area of rectangle is "+ area) ;
    }
    public void Perimeter(float l, float b) {
        double peri = 2 * (l + b) ;
        System.out.println("Perimeter of rectangle is "+ peri) ;
    }
    public void Diagonal(float l, float b) {
        double diag ;
        diag = Math.sqrt((l*l)+(b*b)) ;
        System.out.println("Diagonal: "+ diag) ;
    }
}
```

```

public static void main(String[ ] args) throws IOException {
    Ex17 obj1 = new Ex17( );
    BufferedReader br = new BufferedReader( new InputStreamReader(System.in) );
    float l, b ;
    System.out.println("Enter length & breadth of rectangle : ");
    l = Float.parseFloat( br.readLine( ) );
    b= Float.parseFloat( br.readLine( ) );
    System.out.println("\tMenu");
    System.out.println("1. Area ");
    System.out.println("2. Perimeter ");
    System.out.println("3. Diagonal ");
    System.out.println("Enter your choice 1, 2, 3 : ");
    int ch;
    ch = Integer.parseInt(br.readLine( ) );
    if ( ch == 1)      obj1.Area( l , b );
    else if ( ch == 2)  obj1.Perimeter( l , b );
    else if ( ch == 3)  obj1.Diagonal( l , b );
    else System.out.println("Wrong Choice!");
}
}

```

Enter length & breadth of rectangle :

5.3f

4.3f

Menu

1. Area
2. Perimeter
3. Diagonal

Enter your choice 1 / 2 / 3 :

1

Area of rectangle is 22.790000915527344

6.9 Nested Loops

A loop may contain another loop in its body. This form of a loop is called *nested loop*. But in a nested loop, the inner loop must terminate before the outer loop.

The following is an example of a nested loop :

```

:
for ( i = 1 ; i < 5 ; ++i ) {
    System.out.println( );
    for ( j = 1 ; j <= i ; ++j )
        System.out.print( " * " );
}
//outer loop
//inner loop

```

The output produced is

```

*
* *
* * *
* * * *

```

The inner **for** loop is executed for each value of *i*. The variable **i** takes values 1, 2, 3 and 4. The inner loop is executed once for *i*=1 according to condition *j*=*i* (*i.e.*, 1 ≤ 1 means once), twice for *i*=2, thrice for *i*=3 and four times for *i*=4.

6.10 Comparison of Loops

Though Java loops can be used in almost all situations yet there are some situations where one loop fits better than the other.

The **for** loop is appropriate when you know in advance how many times the loop will be executed. The other two loops **while** and **do-while** loops are more suitable in the situations where it is not known before-hand when the loop will terminate. The **while** should be preferred when you may not want to execute the loop body even once (in case the test condition is false), and the **do-while** loop should be preferred when you're sure you want to execute the loop body at least once.

These are not hard-and-fast rules. The choice of a loop should actually depend upon the point that the loop should make your program the clearest to understand and easiest to follow.

6.11 Jump Statements

The jump statements unconditionally transfer program control within a function. Java has three statements that perform an unconditional branch : **return**, **break**, and **continue**. Of these, you may use **return** anywhere in the program whereas **break** and **continue** are used inside smallest enclosings like loops etc. In addition to the above three, Java provides a standard library function **System.exit()** that helps you break out of a program.

The **return** statement is used to return from a function. This statement has been explained in details under chapter '*Functions*'. Under this section we will deal with the rest of the jump statements : **break** and **continue**.

6.11.1 The **break** Statement

The **break** statement enables a program to skip over part of the code. A **break** statement terminates the smallest enclosing **while**, **do-while**, **for** or **switch** statement. Execution resumes at the statement immediately following the body of the terminated statement. The following figure (Fig. 6.6) explains the working of a **break** statement :

Note A **break** statement skips the rest of the loop and jumps over to the statement following the loop.

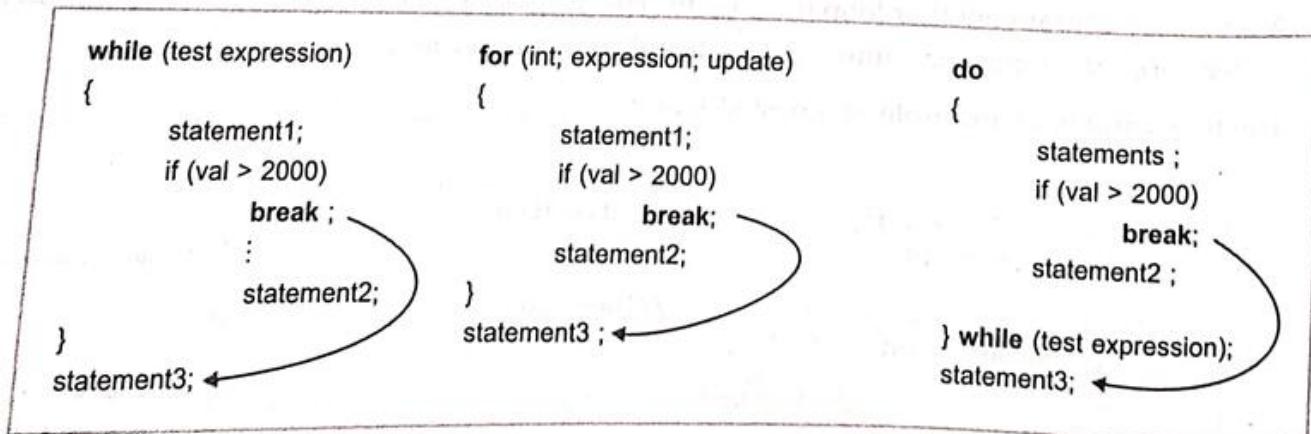


Figure 6.6 The working of a **break** statement.

The following code fragment gives you an example of a break statement:

```
public void Quotient(int a, int b) {
    /* values of integers a, b are received through parameters */
    int i, c;
    for (i = 0; i < 20; ++i) {
        if (b == 0)
            break;
        else
            c = a/b;
        System.out.println ("Quotient = " + c);
    }
    System.out.println ("Program over !");
}
```

The above code fragment inputs two numbers. If the number b is zero, the loop is immediately terminated and just a message *Program Over!* gets displayed otherwise the numbers are repeatedly input and their quotients are displayed.

If a break statement appears in a nested-loop structure, then it causes an exit from only the very loop it appears in. For example,

```
/*
 * here character ch1 is available, which is received as parameter */
for (i = 0; i < 10; ++i) {
    j = 0;
    char ch2 = ch1;
    System.out.println(ch1);
    for (;;) {
        System.out.print (ch2);
        j++;
        if (j == 10)
            break;
    }
    System.out.println("-----");
}
```

The above code fragment inputs a character and prints it 10 times. The inner loop has an infinite loop structure but the break statement terminates it as soon as j becomes 10 and the control comes to the statement following the inner loop which prints a line of dashes. A break used in a switch statement will affect only that switch i.e., it will terminate only the very switch it appears in. It does not affect any loop the switch happens to be in.

TIP One can tell whether a loop has executed a break statement by examining the loop variable.

A loop variable declared before the loop remains in scope even after the loop is terminated either on maturation or prematurely. By examining the value of this loop variable, it can be told whether the loop has been terminated prematurely.

The following code fragment illustrates it :

```

:
int i ;
for ( i = 0 ; i < 100 ; ++i) {
:
if (ch == 'q') break ;
:
}
if ( i < 100)
    System.out.println ("Loop has ended prematurely") ;
else
    System.out.println ("Normal termination has taken place") ;

```

If the *test expression* of the loop is still true even after the termination of the loop, the loop has executed a **break** statement.

6.11.2 The continue Statement

The **continue** is another jump statement like the **break** statement as both the statements skip over a part of the code. But the **continue** statement is somewhat different from **break**. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between. The following figure (Fig. 6.7) explains the working of **continue** statement.

Note The **continue** statement skips the rest of the loop statements and causes the next iteration of the loop.

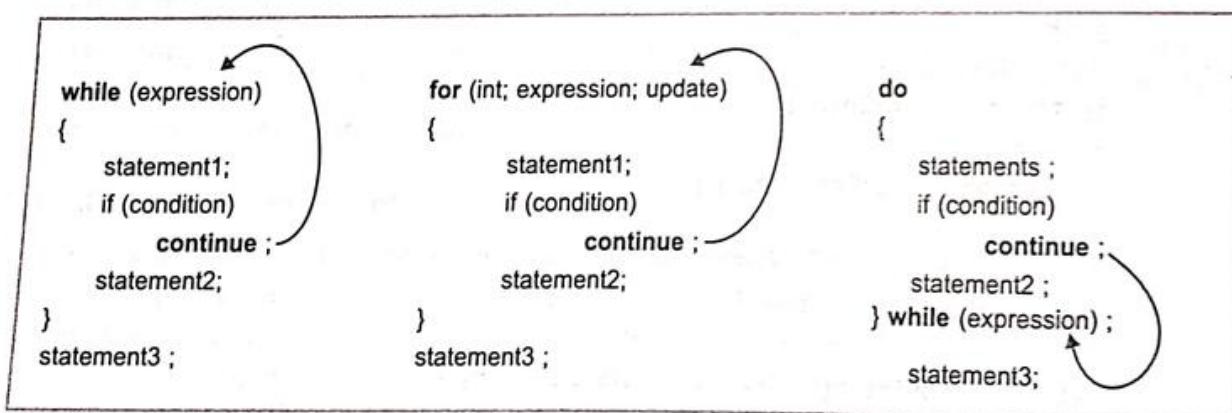


Figure 6.7 The working of a **continue** statement.

For the **for** loop, **continue** causes the next iteration by updating the variable and then causing the test-expression's evaluation. For the **while** and **do-while** loops, the program control passes to the conditional tests. The following code-fragment gives you an example of **continue** statement :

```

:
int a, b, c, i ;
for ( i = 0 ; i < 20 ; ++i) {
    System.out.println Enter 2 numbers" ;
    // two numbers read here ;

```

```

if (b == 0) {
    System.out.println ("The denominator cannot be zero. Enter again");
    continue;
}
else {
    c = a/b ;
    System.out.println ( "Quotient =" + c)
}
}

```

Sometimes you need to abandon iteration of a loop prematurely. Both the statements **break** and **continue** can help in that but in different situations.

TIP

Do not confuse the **break** and **continue** statements

A **break** statement inside a loop will abort the loop and transfer control to the statement following the loop. A **continue** statement will just abandon the current iteration and let the loop start the next iteration.

6.11.3 Labels and Branching Statements

The **break** and **continue** statements can also be used with labels. Let us learn about labels and labelled jump statements.

Since we often nest loops, Java also provides a mechanism for exiting nested loops. You can label any of the loops or switches with *label : statement* and then use **break label** to break out of that statement. For instance, consider the following code fragment that uses labelled loops and labelled jump statements.

```

outer:   ← This label indicates that following
for ( i = 0 ; i < N ; ++i )
{
    inner:   ← This label indicates that following
    for ( j = 0 ; j < M ; ++j ) {
        if ( i > j ) {
            System.out.println ( i + ">" + j );
            break outer;
        }
        else if ( i == j )
            break inner;
    } // end of inner for loop
    // Here's where you go if you break out of the inner loop
} // end of outer for loop
// Here's where you go if you break out of the outer loop

```

Similarly, **continue** can take a label, and returns to the next iteration of the labelled loop.

Labels

As you can see in above example code, *statements* can be labelled. *Labels* are typically used on *blocks* and *loops*. Labelled *blocks* are useful with **break** and **continue**.

The syntax for labelling a statement is :

label: statement

i.e., you label a statement simply by placing a legal Java identifier (the label) followed by a colon (:) before the statement.

Labelled break Statement

break [label] ;

break causes the flow of control to jump to the statement immediately following the current statement. **break label** causes flow of control to break out of the containing statement which must be labelled **label**. In the above given example code fragment,

break outer ;

statement breaks out of **outer loop** rather than innermost loop, which is the default functionality of a **break** statement, in case the **label** is missing.

Labelled continue Statement

A **continue** statement skips the current pass of a loop's body. A continue statement can also be labelled as per following syntax :

continue [label] ;

Statement **continue label** applies the **continue** to the *named loop* (or labelled loop) instead of the innermost loop (the default functionality when label is missing). A labelled continue will break out of any inner loops on its way to the next iteration of the named loop.



Let Us Revise

- ❖ The statements that allow a set of instructions to be performed repeatedly are **iteration statements**.
- ❖ The iteration statements are also called **loops** or **looping statements**.
- ❖ Java provides three loops : **for**, **while** and **do-while**.
- ❖ Four elements control a loop : initialization expression(s), test expression, update expression and loop-body.
- ❖ Initialization expression(s) initialise(s) the loop variables in the beginning of the loop.
- ❖ Test-expression's truth value decides whether the loop will be executed (if test expression is true) or not (if test expression is false).
- ❖ The entry-controlled loops impose control at the time of entry into the loop by testing the test-expression before entering into a loop. The **for** and **while** loops are entry-controlled loops.
- ❖ The exit-controlled loops impose control at the time of exit from the loop by testing the test-expression before exiting from the loop. The **do-while** is an exit-controlled loop.
- ❖ The update expression(s) update(s) the values of loop variables after every iteration of the loop.
- ❖ The loop-body contains statements to be executed repeatedly.
- ❖ If the loop-body of a loop contains an empty statement i.e., null statement, the loop is called an **empty loop**.
- ❖ Java allows the declaration of variables within the loop's control elements also.
- ❖ The **while** loop evaluates a test-expression before allowing entry into the loop.
- ❖ A **while** loop can also be infinite loop, if its loop control variable is not updated within its body.
- ❖ A **while** loop can also be an empty loop, if it contains just a null statement in its body.

Note

Two or more statements cannot have the same label name under the same method.

- ❖ The **do-while** is executed at least once always as it evaluates the test expression at the end of the loop.
- ❖ The loops can be nested also, if a loop contains another loop inside its body.
- ❖ In nested loops, the inner loop must terminate before the outer loop terminates.
- ❖ The statements that facilitate the unconditional transfer of program control are called **Jump statements**.
- ❖ Java provides three jump statements : **return**, **break** and **continue**.
- ❖ The **return** statement is used to return from a function.
- ❖ A **break** statement can appear in any of the loops and a **switch** statement. Whichever statement it appears in, it causes the termination of the statement there and then and the control passes over to the statement following the statement containing **break**.
- ❖ In nested structure, a **break** terminates the very statement it appears in.
- ❖ The **continue** statement abandons the current iteration of the loop by skipping over the rest of the statements in the loop-body. It immediately transfers control to the evaluation of the test-expression of the loop for the next iteration of the loop.
- ❖ Loops can be labelled also.
- ❖ The **break** and **continue** can be labelled also, to break out or continue to next iteration of the specified loop.

Solved Problems

1. Given the value of a variable, write a statement, without using if construct, which will produce the absolute value of the variable.

Solution. $x = x < 0 ? -x : x ;$

2. What is wrong with the following code ?

```
switch(x)  {
    case 1 :
        n1 = 10 ;
        n2 = 20 ;
    case 2 :
        n3 = 30 ;
        break ;
        n4 = 40 ;
}
```

Solution. $n = 40$; is unreachable.

3. What is the problem with the following snippet ?

```
class Q3  {
    public void Test( )  {
        int i = 5, j = 10 ;
        if((i < j) || (i = 10 ))
            System.out.println("OK");
        System.out.println("NOT OK");
    }
}
```

Solution. ($i = 10$) used with if is the problem.

4. Show the output of the following code :

```
int a = 5, b = 10 ;
if(a > 5)
```

```

        if(b > 5) {
            System.out.println( " b is " +b );
        }
        else
            System.out.println( "a is " + a );
    
```

Solution. a is 5.

5. State the output of the following code :

```

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

Solution. No output.

6. How is the if...else if combination more general than a switch statement ?

Solution. The switch statement must be controlled by a single integer control variable, and each case section must correspond to a single constant value for the variable. The if...else if combination allows any kind of condition after each if.

7. What is a "fall through" ?

Solution. The term "fall through" refers to the way the switch statement executes its various case sections. Every statement that follows the selected case section will be executed unless a break statement is encountered.

8. Consider the following two code fragments for counting spaces and newlines :

<pre>// version 1 : if (ch == ' ') spaces++; if (ch == '\n') newlines++; </pre>	<pre>// version 2 : if (ch == ' ') spaces++; else if (ch == '\n') newlines++; </pre>
---	--

What advantages, if any, does the second version have over the first ?

Solution. Version1 uses two separate if statements. Thus, two times the conditions are tested whereas the version2 uses an if-else construct. If the first condition is true in version2, the second condition is never tested in contrast to version1 where both conditions are always tested irrespective of whether the first condition is true or false.

Therefore, the version1 takes more processing time as compared to version2 (in case the first condition is true). The version2 is more efficient compared to version1.

9. What will be the output of following two code fragments :

<pre>// version1 i = j = 10 ; if (a < 100) if (b > 50) ++i ; </pre>	<pre>// version2 i = j = 10 ; if (a < 100) { if (b > 50) ++i ; } </pre>
---	---

```

        else
            ++j;
    System.out.println ("i =" + i)
    System.out.println ("j =" + j);
}
else
    ++j;
System.out.println ("i =" + i);
System.out.println ("j =" + j);

```

if the input given is shown below : (i) a = 30, b = 30 (ii) a = 60, b = 70

Solution. case (i) when a = 30, b = 30

Version 1's output will be as follows :

```
i = 10
j = 11
```

case (ii) when a = 60, b = 70

Version 1's output will be as follows :

```
i = 11
i = 10
```

Version 2's output will be as follows :

```
i = 10
j = 10
```

Version 2's output will be as follows :

```
i = 11
j = 10
```

10. Identify the error(s) in the following code fragment :

```

:           // declaration and initialization of stream object
char ch;
int vowels = 0, others = 0 ;
System.out.println ("Enter character :");
ch = (char) (in.read( ) );
switch (ch) {
    case 'a' :
    case 'A' :
    case 'e' :
    case 'E' :
    case 'i' :
    case 'I' :
    case 'o' :
    case 'O' :
    case 'u' :
    case 'U' : + + vowels ;
                break ;
    default : + + others ;
}
:
```

Solution. The errors in the above code fragment are :

1. An uninitialized variable ch is being used in while's test expression. The variable ch must have a value before it is used in an expression.
 2. The switch statement's two case constants are identical case 'i' and case 'I'. The case constants of same switch must have different values.
11. What will be the output of following code fragment if the input is
 (i) a (ii) c (iii) d (iv) h (v) b ?

```

:
ch = character.readChar( );
```

```

switch (ch) {
    case 'a' : System.out.println ("It is a.");
    case 'b' : System.out.println ("It is b.");
    case 'c' : System.out.println ("It is c.");
        break;
    case 'd' : System.out.println ("It is d.");
        break;
    default : System.out.println ("Not a b c d.")
}
}

```

Solution. (i) When input is *a*, the output will be as follows :

It is *a*.
It is *b*.
It is *c*.

(ii) When input is *c*, the output will be as follows : It is *c*.

(iii) When input is *d*, the output will be : It is *d*.

(iv) When input is *h*, the output will be : Not *abcd*.

(v) When input is *b*, the output will be : It is *b*
It is *c*.

12. Give the output of the following code :

```

int m = 100 ;
while(m > 0) {           // true is Boolean true
    if(m < 10)
        break ;
    m = m - 10 ;
}
System.out.println("m is" + m) ;

```

Solution. m is 0.

13. Give the output of the following code :

```

int m = 100 ;
while(true) {
    if(m < 10)
        continue ;
    m = m - 10 ;
}
System.out.println("m is" + m) ;

```

Solution. No output ; Infinite loop

14. What does a break statement do ?

Solution. A break statement terminates the current loop and proceeds to the first statement that follows that loop. For example, the break statement in the following loop

```

for(int d = 2 ; d < n ; d++) {
    isNotPrime = (n%d == 0) ? 1 : 0 ;
    if(isNotPrime == 0) break ;
}

```

stops the loop and executes the next statement that follows it.

15. What does a labelled break statement do? When would you use a labelled break statement instead of an unlabelled one?

Solution. A labelled break statement terminates the current loop and proceeds to the first statement that follows the loop that is labelled by the identifier that follows the keyword break. For example, the statement

```
if(i == 1 && j == 2 && k == 0)
    break resume;
```

contains a labelled break statement that terminates both the inner loop and the middle loop containing it.

- 16 Predict the output from the following program. Then run it to confirm your prediction:

```
class q16 {
    public static void main( String args[ ] ) {
        int count = 0;
        for(int i = 0 ; i < 3 ; i++)
            resume:
            for(int j = 0 ; j < 4 ; j++)
                for(int k = 0 ; k < 5; k++ ) {
                    ++count;
                    if(i == 1 && j == 2 && k == 3) break resume;
                }
            System.out.println("\t count = " + count);
    }
}
```

Solution. count = 54

- 17 Predict the output from the following modification of the program from Q. 16. Then run it to confirm your prediction:

```
class q17 {
    public static void main( String args[ ] ) {
        int count = 0;
        for(int i = 0 ; i < 3 ; i++) {
            resume:
            for(int j = 0 ; j < 4 ; j++)
                for(int k = 0 ; k < 5 ; k++ ) {
                    ++count;
                    if(i == 1 && j == 2 && k == 3) break resume;
                }
            System.out.println("\t count = " + count);
        }
    }
}
```

Solution. count = 20
count = 34
count = 54

- 18 Given the following for loop:

```
:
final int SZ = 25;
for (int i = 0, sum = 0 ; i < SZ ; i++)
```

228

```

    sum += i;
    System.out.println (sum);
    :

```

Write equivalent while loop for the above code.

Solution.

```

final int SZ = 25;
int i = 0, sum = 0;
while (i < SZ) {
    sum += i;
    i++;
}
System.out.println (sum);
:

```

19 What are the outputs of following two code fragments ? Justify your answer.

```

// version 1                                // version 2
int f = 1, i = 2;                          int f = 1, i = 2;
while (++i < 5)                            do {
    f *= i;                                f *= i;
}                                         } while (++i < 5);
System.out.println (f);                     System.out.println (f);
:

```

Solution. The output of version 1 will be 12.

The reason for it is that a **while** evaluates test expression before entering into the loop. Thus the value of i for the first iteration becomes 3 (because of **++i** in test expression) and the loop gets executed twice for two values of i, 3 & 4. The f gets the value of $f * 3 * 4$ which is 12.

The output of version 2 will be 24.

The reason for it is that a **do-while** first executes the loop and then evaluates the test expression before exiting the loop. Thus the value of i for first iteration of loop remains 2 and the loop gets executed thrice for three values of i, 2, 3 & 4. The f gets the value of $f * 2 * 3 * 4$ which is 24.

20. Consider the following code fragment :

```

int line = 0;
char ch = (char) System.in.read();
while (ch != '&') {
    ch = (char) System.in.read();
    if (ch == 'Q')
        break;
    if (ch != '\n')
        continue;
    line++;
}

```

Rewrite this code without using **break** or **continue**.

Solution.

```

int line = 0 ;
char ch ;
while (ch != '&' && ch != 'Q') {
    ch = (char) System.in.read() ;
    if (ch == '\n')
        line ++ ;
}

```

21. Write a Java program to find out whether a year (entered in 4-digit number representing it) is a leap year.

Solution.

```

class LeapYr {
    public void Test ( int year ) {
        if ( year %100 == 0 ) {
            if (year % 400 == 0)
                System.out.println (year + " is a century year and a Leap year");
            else
                System.out.println (year + " is a century year but not a leap year");
        }
        else if (year % 4 == 0)
            System.out.println (year + " is a leap year");
        else
            System.out.println (year + " is not a leap year");
    }
    static public void main(String args[ ]) {
        LeapYr obj1 = new LeapYr();
        obj1.Test(1900);
    }
}

```

22. Given three numbers A, B and C, write a program to write their values in descending order. For example, if A = 7, B = 4, and C = 10, your program should print out : 10, 7, 4.

Solution.

```

class Arrange {
    public void arrangeDesc( int a, int b, int c) {
        int big = 0, big2 = 0, big3 = 0;
        big = a;
        if(b > big)
            big = b;
        if(c > big)
            big = c;
        if(a == big) {
            if (b > c) {
                big2 = b;      big3 = c;
            }
            else {
                big2 = c;      big3 = b;
            }
        }
    }
}

```

```

        else if(b == big) {
            if (a>c) {
                big2 = a;           big3 = c;
            }
            else {
                big2 = c;           big3 = a;
            }
        }
        else if(c == big) {
            if (a>b) {
                big2 = a;           big3 = b;
            }
            else {
                big2 = b;           big3 = a;
            }
        }
    }
    System.out.println ("Numbers in descending order : " + big + ","
                        + big2 + "," + big3);
}
public static void main(String args[ ]) {
    Arrange object1 = new Arrange();
    object1.arrangeDesc(7, 10, 4);
}
}

```

23. During a special sale at a store, a 10% discount is taken on purchases over ₹ 1000/- . Write a program that asks for the amount of purchases, then calculates the discounted price. The purchase amount will be input in ₹ :

Enter amount of purchases :

2000

Discounted price : 1800

Use integer arithmetic throughout the program.

Solution.

```

import java.io.*;
class Sale {
    public void Discount( float purchaseAmount ) {
        float discount = 0;
        if (purchaseAmount > 1000)
            discount = purchaseAmount / 10;
        System.out.println ("Discounted price : " + (purchaseAmount - discount));
    }
    public static void main(String args[ ]) {
        Sale acc1 = new Sale();
        float saleamt;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        try {
            System.out.println("Enter amount of purchases : ");
            saleamt = Float.parseFloat(br.readLine());
        }
    }
}

```

```
    acc1.Discount(saleamt);  
}  
catch(Exception e) {  
    System.out.println(e);  
}
```

24. Write a short program to input a digit and print it in words.

Explanation

```

import java.io.*;
class Translate {
    public void NumInWords( int num) {
        int temp, digit;
        String result = " ";
        temp = num;
        do {
            digit = temp%10;
            result = getWordForDigit (digit) + " " + result;
            temp = temp / 10;
        } while (temp != 0);
        System.out.println ("The number " + num + " in words is : \n" + result);
    }
    public static String getWordForDigit (int digit) {
        String ans= " ";
        switch (digit) {
            case 0: ans= "Zero";
            break;
            case 1: ans= "One";
            break;
            case 2: ans= "Two";
            break;
            case 3: ans= "Three";
            break;
            case 4: ans= "Four";
            break;
            case 5: ans= "Five";
            break;
            case 6: ans= "Six";
            break;
            case 7: ans= "Seven";
            break;
            case 8: ans= "Eight";
            break;
            case 9: ans= "Nine";
            break;
        }
        return ans;
    }
}

```

```

public static void main(String args[ ] ) {
    Translate number = new Translate( );
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    int num;
    try {
        System.out.println("Enter an integer : ");
        num = Integer.parseInt(br.readLine( ));
        number.NumInWords(num);
    }
    catch(Exception e) {
        System.out.println(e);
    }
}

```

25. Write a short program to find whether the given character is a digit or a letter.

Solution.

```

import java.io.*;
class charType {
    public void DigOrLetter ( char ch ) {
        if ( Character.isDigit (ch) )
            System.out.println ("The character represents a digit");
        else if ( Character.isLetter (ch) )
            System.out.println ("The character represents a letter");
        else
            System.out.println ("The character represents neither a digit nor a letter");
    }
    public static void main(String args[ ] ) {
        charType obj1 = new charType();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        char ch;
        try {
            System.out.println("Enter a character : ");
            ch = (char) (br.read( ) );
            obj1.DigOrLetter(ch);
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}

```

26. Write a java program to check whether the given number is palindrome or not.

Solution.

```

import java.io.*;
class Test {
    public void PalindromeNumber(int num) {
        int n = 0, digit, rev = 0 ;
        n = num ;

```

```

do {
    digit = num%10 ;
    rev = (rev*10) + digit ;
    num = num/10 ;
} while(num != 0) ;
System.out.println("The reverse of the number is :" + rev) ;
if(n == rev)
    System.out.println("The number is palindrome.") ;
else
    System.out.println("The number is not palindrome.") ;
}

public static void main( String args[ ] ) {
    Test object1 = new Test( ) ;
    BufferedReader br = new BufferedReader ( new InputStreamReader(System.in)) ;
    int num;
    try {
        System.out.println("Enter a number : ") ;
        num = Integer.parseInt(br.readLine( )) ;
        object1.PalindromeNumber(num) ;
    }
    catch(Exception e) {
        System.out.println(e) ;
    }
}

```

27. Write a java program to generate divisors of an integer.

Solution.

```

import java.io.*;
class Generate {
    public void Divisors(int num) {
        int i = 0 ;
        System.out.println("Divisors of given number "+num+" are : ") ;
        for ( i = 1 ; i <= num/2 ; i++) {
            if (num%i == 0)
                System.out.println(i) ;
        }
    }
}

public static void main( String args [ ] ) throws IOException {
    Generate object1 = new Generate( ) ;
    BufferedReader br = new BufferedReader( new InputStreamReader ( System.in ) ) ;
    int num ;
    System.out.println("Enter a number : ") ;
    num = Integer.parseInt( br.readLine( )) ;
    object1.Divisors(num) ;
}

```

28. Write a java program to find whether a given number is odd or even or prime.

Solution.

```

import java.io.*;
class Test {
    public void OddEvenPrime(int num) {
        int ch = 0, i, flag = 0;
        System.out.print("The number "+num+" is ");
        if (num%2 == 0)
            System.out.print("Even");
        else
            System.out.print("Odd");
        for (i = 2; i <= num/2; ++i)
            if(num% i == 0) {
                flag = 1;
                break;
            }
        if (flag == 0)
            System.out.println(" and Prime.");
        else
            System.out.println(" but Not Prime.");
    }
    public static void main(String args[] ) throws IOException {
        Test object1 = new Test();
        int num;
        BufferedReader br = new BufferedReader(new InputStreamReader( System.in));
        System.out.println("Enter a number ");
        num = Integer.parseInt(br.readLine());
        object1.OddEvenPrime(num);
    }
}

```

29. Write a java program to compute cosine series : $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots - \frac{x^n}{n!}$

Solution.

```

import java.io.*;
class Series {
    public void CosSeries(double x) {
        int n = 20, i;
        double t = 0, sum, y = x;
        x = x * 3.1412/180; // converted into radians
        t = t + 1;
        sum = 1;
        for(i = 1; i <= n; ++i) {
            t = t * Math.pow((double)(-1), (double)(2*i - 1)) * x*x/(2*i*(2*i - 1));
            sum = sum + t;
        }
        System.out.println("Cos( "+y+" ) = " + sum);
    }
}

```

```

public static void main(String args[ ] ) throws IOException {
    Series object1 = new Series( );
    double x;
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter value of x for Cos(x) series : ");
    x = Double.parseDouble(br.readLine());
    object1.CosSeries(x);
}
}

```

30. Write a java program to print table of a given number.

Solution.

```

import java.io.*;
class Number {
    public void Table(int num) {
        for( int i = 1 ; i < 11 ; ++i)
            System.out.println(num + " * " + i + " = " + num * i);
    }
    public static void main(String args[ ] ) throws IOException {
        Number object1 = new Number();
        int num;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter a number ");
        num = Integer.parseInt(br.readLine());
        object1.Table(num);
    }
}

```

31. Write a java program to print fibonacci series i.e., 0 1 1 2 3 5 8 . . . The number of terms required, is to be passed as parameter.

Solution.

```

class FibSeries {
    public void Fibo(int n) {
        int i;
        long first, second, third;
        first = 0; second = 1;
        System.out.println("\nFibonacci Series -> ");
        System.out.print(first + "\t" + second);
        for(i = 2 ; i < n ; i++) {
            third = first + second;
            System.out.print("\t" + third);
            first = second;
            second = third;
        }
    }
    public static void main( String args[ ] ) {
        FibSeries object1 = new FibSeries();
        object1.Fibo(12);
    }
}

```

32. Write a Java program to print every integer between 1 and n divisible by m . Also report whether the number that is divisible by m is even or odd.

Solution.

```

class Test {
    public void PrintNDivide( int n, int m ) {
        /* n is the limiting number, m is the divisor */
        System.out.println ( " The numbers between the range 1 - " + n
            + " that are divisible by " + m + " are: " );
        for ( int i=1; i<=n; i++ )
            if ( i % m == 0 ) {
                System.out.print ( i );
                if ( i%2 == 0 )
                    System.out.println ( " : even" );
                else
                    System.out.println ( " : odd" );
            }
    }
    public static void main( String args [ ] ) {
        Test object1 = new Test( );
        object1.PrintNDivide(30, 3); // to print numbers divisible by 3 in 1-30.
    }
}

```



Glossary

Fall through Execution of multiple cases after matching takes place in a switch statement.

Selection Statement Statement that allows to choose a set-of-instructions for execution depending upon an expression.

Statement Instruction given to the computer to perform any kind of action.

Infinite Loop A loop that never ends.

Iteration Statement Statement that allows a set of instructions to be performed repeatedly.

Jump Statement Statement that unconditionally transfers program control within a function.

Looping Statement Iteration statement. Also called a loop.

Nested Loop A loop that contains another loop inside its body.



Assignments

TYPE A : VERY SHORT/SHORT ANSWER QUESTIONS

- What is the significance of a null statement ?
- What are the three constructs that govern statement flow ?
- What is a selection statement ? Which selection statements does Java provide ? Write one advantage and one disadvantage of using `? :` in place of an `if`.
- Can a conditional operator replace an `if` statement always ? Suggest a way to override the default dangling-else matching.

5. In a nested-if, how does the default matching of dangling-else take place ?
6. One out of several different alternatives can be selected with the help of which statement ?
7. What is the significance of a **break** statement in a switch statement ?
8. The case constants in a switch statement can be of any data type. (T/F) ?
9. Write one limitation and one advantage of a switch statement.
11. What is the effect of absence of **break** in a switch statement ?
11. The case labels in a switch can have identical values. (T/F) ?
12. What is the significance of default clause in a switch statement ?
13. What are iteration statements ? Name the iteration statements provided by Java.
14. Which elements are needed to control a loop ?
15. What is meant by an entry-controlled loop ? Which Java loops are entry-controlled ?
16. What is meant by an exit-controlled loop ? Which Java loops are exit-controlled ?
17. The update expression in a for loop can decrement the loop variable. T/F ?
18. The initialization expression of a for loop must be followed by a semicolon. T/F ?
19. The update expression of a for loop must be followed by a semicolon. T/F ?
20. Write a for loop that displays the numbers from 51 to 60.
21. Write a for loop that displays the numbers from 10 to 1 i.e., 10, 9, 8.... 3,2,1
22. Which expressions are optional in a for loop ? When are empty loops useful ?
23. What is meant by a variable's scope ?
24. What is the difference between a while and do-while loop ?
25. How is break statement different from a labelled break statement ?
26. When are labelled loops useful ?
27. A code displaying a menu must be executed at least once. Which loop is most suitable for it ?

TYPE B : SHORT ANSWER QUESTIONS

1. What is the problem of dangling-else ? When does it arise ? What is the default dangling-else matching and how can it be overridden ?
2. Compare an if and a ?: operator.
3. Given the following code fragment :

```

if (a == 0)
    System.out.println ("Zero") ;
if (a == 1)
    System.out.println ("One") ;
if (a == 2)
    System.out.println ("Two") ;
if (a == 3)
    System.out.println ("Three") ;

```

Write an alternative code (using if) that saves on number of comparisons.

4. Rewrite the following fragment using switch :

```

if (ch == 'E')
    eastern++ ;
if (ch == 'W')
    western++ ;

```

```

if (ch == 'N')
    northern++ ;
if (ch == 'S')
    southern++ ;
else
    unknown++ ;

```

5. Write the syntax and purpose of a **switch** statement.
6. When does an **if** statement prove more advantageous over a **switch** statement ?
7. When does a **switch** statement prove more advantageous over an **if** statement ?
8. Why is it suggested to put a **break** statement after the last case statement in a **switch** even though it is not needed syntactically ?
9. Rewrite the code given in question 3 using **switch**.

10. Rewrite the following set of **if-else** statements in terms of **switch-case** statement :

```

(a) char code ;
    code = Character.readChar( ) ;
    if (code == 'A')
        System.out.println ("Accountant") ;
    else if (code == 'C' || code == 'G')
        System.out.println ("Grade IV") ;
    else if (code == 'F')
        System.out.println ("Financial Advisor") ;

```

```

(b) int inputnum, calcval ;
    if (inputnum == 5)
        {
            calcval = inputnum * 25 - 20 ;
            System.out.println (inputnum + calcval) ;
        }
    else if (inputnum == 10)
        {
            calcval = inputnum * 25 - 20 ;
            System.out.println (calcval - inputnum) ;
        }

```

11. Rewrite the following after removing an extra statements

```

;
public void QuadRoots( )
{
    float a, b, c, Disc, X ;
    Disc = b * b ;
    Disc = Disc - 4 * a * c ;
    if (Disc > 0)    {
        System.out.println ("Roots are real and") ;
        System.out.println ("unequal") ;
    }
    else if (Disc == 0)  {
        System.out.println ("Roots are") ;
        System.out.println ("Real & equal") ;
    }
    else if (Disc < 0) System.out.println ("No real roots") ;
}

```

12. How many times are the following loops executed?

(a) `x = 5 ; y = 50 ;`

`while(x <= y) {`

`x = y/x ;`

`.....`

`}`

(b) `int m = 10, n = 7 ;`

`while(m % n >= 0) {`

`.....`

`m = m + 1 ;`

`n = n + 2 ;`

`..... }`

13. Given the following code fragment :

`i = 2 ;`

`start :`

`System.out.println (i) ;`

`i += 2 ;`

`if (i < 51) goto start ;`

`System.out.println ("Thank you") ;`

Rewrite the above code using a `while` loop.

14. Given the following code fragment :

`i = 100;`

`while (i > 0)`

`System.out.println (i--)`

`System.out.println ("Thank you") ;`

Rewrite the above code using a `goto` statement.

15. Rewrite following code using while loop

`int sum = 0 ;`

`for (int i = 1 ; i <= 5 ; ++i)`

`sum = sum + c ;`

`}`

16. Rewrite following while loop into a for loop

`int stripes = 0 ;`

`while (stripes <= 13) {`

`if (stripes %2 == 2)`

`{`

`System.out.println("colour code Red") ;`

`}`

`else {`

`System.out.println("colour code Blue") ;`

`}`

`System.out.println("New stripe") ;`

`stripes = stripes + 1 ;`

`}`

17. Rewrite following code using either while or do-while loop or both loops.

`for(int i = 1 ; i < 4 ; ++i) {`

`for(int j = 3 ; j > 0 ; --j) {`

`System.out.print("###..") ;`

`}`

`System.out.println() ;`

`}`

OUTPUT AND ERROR QUESTIONS

18. Find the output of the following code fragments ?

(a)

```
int s = 14 ;
if ( s < 20)
    System.out.print("under") ;
else
    System.out.print("Over") ;
System.out.println("the limit.") ;
```

(b)

```
int s = 14 ;
if(s < 20)
    System.outprint("under") ;
else {
    System.out.print("over") ;
    System.out.println("the limit");
}
```

(c)

```
int s = 94 ;
if (s < 20) {
    System.out.print("under") ;
}
else {
    System.out.print("over") ;
}
System.out.println("the limit") ;
```

19. What will be the output of following code fragment when the input is

- (a) 'A' (b) 'C' (c) 'D' (d) 'F' ?

```
ch = Character.readChar( ) ;
switch (ch)
case 'A' : System.out.println ("Grade A") ;
case 'B' : System.out.println ("Grade B") ;
case 'C' : System.out.println ("Grade C") ;
    break;
case 'D' : System.out.println ("Grade D") ;
default : System.out.println "Grade F";
```

20.

```
int n = 0 ; int j = 7 ; int i = 3 ;
if (i > 2) {
    n = 1 ;
    if (j > 4)
        n = 2 ;
    else
        n = 3 ;
    else {
        n = 4 ;
        if (j%2 >= 2)
            n = 5 ;
        else
            n = 6 ;
}
System.out.println(n) ;
```

21. Predict the output of the following code fragments :

(a) int i, j, n ;
 n = 0 ; i = 1 ;
 do {
 n++ ; i++ ;
 } while (i <= 5) ;

(b) int i = 1, j = 0, n = 0 ;
 while (i < 4){
 for(j = 1 ; j <= i ; j++) {
 n += 1 ;
 }
 i = i + 1 ;
 }
 System.out.println(n) ;

(c) int i = 3, n = 0 ;
 while (i < 4){
 n++ ; i-- ;
 }
 System.out.println(n) ;

(d) int j = 1, s = 0 ;
 while (j < 10) {
 System.out.print(j + "+") ;
 s = s + j ;
 j = j + j % 3 ;
 }
 System.out.println("=" + s) ;

22. Find out errors, if any :

(a) m = 1 ;
 n = 0 ;
 for(; m + n < 19 ; ++n)
 System.out.println("Hello \n") ;
 m = m + 10 ;

(b) while(ctr != 10) {
 ctr = 1 ;
 sum = sum + a ;
 ctr = ctr + 1 ;
}

(c) for(a = 1, a > 10 ; a = a + 1)
 {

 }

23. Identify the possible error(s) in the following code fragment : Discuss the reason(s) of error(s) and correct the code.

```
f = 1 ;
for (int a = 40 ; (a) ; a--)
    f *= a ;
:
s = 0 ;
for (int a = 1 ; a < 40 / a++)
    s += a ;
```

24. Identify the possible error(s) in the following code fragment. Discuss the reason(s) of error(s) and correct the code.

```
while (i < j)
    System.out.println (i * j) ;
    i++ ;
```

25. Identify the possible error(s) in the following code fragment. Discuss the reason(s) of error(s) and correct the code.

```
while (i < j) {
    System.out.println (i * j) ;
    i++ ;
}
```

TYPE C : LONG ANSWER QUESTIONS

1. Write a program to find whether the given character is a digit or a letter.
2. Write a program to input a digit and print it in words.
3. Write a program to read three *double* numbers and print the largest of these.
4. Given three numbers A , B and C , write a program to write their value in an ascending order. For example, if $A = 12$, $B = 10$, and $C = 15$, your program should print out :

Smallest number = 10

Next higher number = 12

Highest number = 15

5. A bank accepts fixed deposits for one year or more and the policy it adopts on interest is as follows :
 - (i) If a deposit is less than Rs. 2000 and for 2 or more years, the interest rate is 5 percent compounded annually.
 - (ii) If a deposit is Rs. 2000 or more but less than 6000 and for 2 or more years, the interest rate is 7 percent compounded annually.
 - (iii) If a deposit is more than Rs. 6000 and is for 1 year or more, the interest is 8 percent compounded annually.
 - (iv) On all deposits for 5 years or more, interest is 10 percent compounded annually.
 - (v) On all other deposits not covered by above conditions, the interest is 3 percent compounded annually.

Given the amount deposited and the number of years, write a program to calculate the money in the customer's account at the end of the specified time.

6. Write a program to find the LCM and GCD of two numbers.
7. Write a Java program to print every integer between 1 and n divisible by m . Also report whether the number that is divisible by m is even or odd.
8. Write a program to print a tribonacci series (series whose first three terms are given and every successive term is sum of previous three terms e.g.,

0 1 2 3 6 11 20 27

9. Write a Java program to read a number and produce its reversed number.
10. Write a Java program to sum the sequence

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

11. Write a program to find the sum of the series

$$x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \frac{x^5}{5!} - \frac{x^6}{6!}$$

12. Write a program to find sum of the series $S = 1 + x + x^2 + \dots + x^n$.

Classes in Java

7.1 Introduction

As you know Java provides you with many built-in datatypes – the primitive datatypes. There are only eight *primitive (or fundamental) datatypes* viz., *byte, short, int, long, float, double, char, boolean*. Java allows you to use data of any of these eight primitive types or you may even create your own datatypes. Any datatype that you create in Java would be created using primitive datatypes. Such data types that are based on primitive or fundamental datatypes, are known as **Composite data types**. Since these data types are defined by user, these are also referred to as **user-defined datatypes**.

This chapter is dedicated to the discussion of composite/user-defined datatypes. Since these data types are created through classes, we shall be exploring how classes form the platform for creating user-defined types.

The data types that are based on fundamental or primitive data types, are known as **Composite Datatypes**. Since these data types are created by users, these are also known as **User-Defined Datatypes**.

In This Chapter

- 7.1 Introduction
- 7.2 Class as Composite Type
- 7.3 Creating and Using Objects
- 7.4 Encapsulation
- 7.5 Visibility Modifiers
- 7.6 Scope and Visibility Rules

7.2 Class as Composite Type

As mentioned before, a composite type is the one that is based on fundamental datatype. A class is a good example of composite datatype. Consider the following class definition:

```
class TypeDemo {
    byte a ;
    int b ;
    float c ;
    char d ;
    public void getData( ) {
        :
    }
    public void display( ) {
        :
    }
}
```

You can see that primitive datatypes have been used for defining this class :

```
byte a ;
int b ;
float c ;
char d ;
```

Once a class is declared, variables of this class type can be declared and created e.g.,

```
TypeDemo obj1 = new TypeDemo( );
```

As you already know that variables of a class type are known as *objects*.

Hence a class satisfies the condition of becoming a composite datatype. Thus, we can say that a *class is a composite, user-defined datatype*. All reference types can be referred to as *composite datatypes*.

If you want to define your own datatype, you can do so by defining them through classes. Sometimes, some logically related elements need to be treated under one unit. For instance, the elements storing a student's information (e.g., *rollno*, *name*, *class*, *marks*, *grade*) need to be processed together under one roof. Similarly, elements keeping a date's information (e.g., *day*, *month*, and *year*) need to be processed together. To handle and serve to such situations, you can create your own datatype through classes.

To create a datatype for processing *dates*, you may write :

```
class Date {
    byte dd, mm, yyyy ;
    public Date( ) {
        dd = 1 ;
        mm = 1 ;
        yyyy = 2006 ;
    }
```

```

public Date(byte d, byte m, byte y) {
    dd = d ;
    mm = m ;
    yyyy = y ;
}
public void display() {
    System.out.println(dd + "/" + mm + "/" + yyyy) ;
}
:
}

```

Once you define this, the **Date** has become a (composite) datatype, whose variables (*i.e.*, objects) you can declare as and when you need it :

```
Date join_date = new Date(13, 2, 2008) ;
```

The above statement creates or *instantiates* a *Date* object namely *join_date* storing date 13/02/2008.

The size of a composite datatype depends upon its constituent member variables. It is generally the total of all sizes of its constituent members.

7.2.1 Class as User-defined Datatype

In Java, you can define one or more user-defined datatypes via the *class* construct. For example, to create a program that behaves like a dog, you can define a class that (minimally) represents a dog :

```

class Dog {
    void bark() {
        System.out.println("Woof.");
    }
}

```

See, this user-defined datatype begins with the keyword *class*, followed by the name for the datatype, in this case **Dog**, followed by the specification of what it is to be a dog between opening and closing curly brackets. This simple example provides no data fields, only the single behavior of barking, as represented by the method *bark*.

User-defined datatype vs. Application

In Java, all functionality is enclosed in classes. But in order for a class to be user-defined datatype, it should act different from that of an application. That is, it should not include *main* method in it. Although you can create instances of classes containing *main* method, they should not be referred to as user-defined datatype. Such classes (containing *main* method) are more analogous to application than a datatype as a datatype's functionality is implemented in another class not in its own class.

Solved problems 4-7, highlight this very difference of the two.

Note An application has a class containing *main* method.

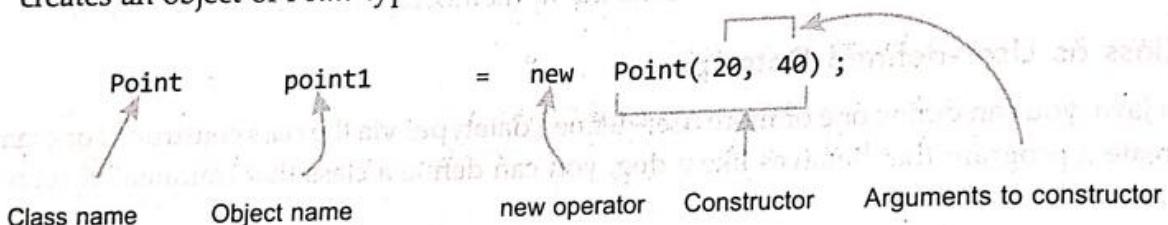
Primitive datatypes vs. User-defined datatype

You can create variables or instances of primitive datatypes as well as of user-defined datatypes. These two categories of datatypes are fundamentally different from one another. Let us see how.

Primitive Datatypes	User-defined Datatypes
(i) These are built-in datatypes. Java provides these datatypes.	These are created by the users.
(ii) The sizes of these datatypes are fixed.	The sizes of these datatypes are variable as their sizes depend upon their constituent members.
(iii) These datatypes are available in all parts of a Java program.	The availability of these datatypes depends upon their scope.

7.3 Creating and Using Objects

As you know that a class provides a blueprint for objects. From this blueprint, objects are created using the `new` operator along with the class constructor e.g., following statement creates an object of *Point* type.



Consider two more statements given below that are also creating objects :

```
Rectangle rect_1 = new Rectangle(50, 100);
Rectangle rect_2 = new Rectangle(point1, 50, 100);
```



see here an argument is an object
of point type created earlier

Each object declaration statement given above has *three* parts :

1. **Declaration.** The code on the left of assignment operator `=`, in all the above given statements are all variable declarations that associate a name with a type. When you create an object, you do not have to declare a variable to refer to it. However, a variable declaration often appears on the same line as the code to create an object.
2. **Instantiation.** `new` is a Java operator that creates the new object (allocates space for it).
3. **Initialization.** The `new` operator is followed by a call to a constructor. For example, `Point(23, 94)` is a call to *Point*'s only constructor. The constructor initializes the new object.

Let us discuss these actions in details :

1. Declaring Object Variable

As you know that a variable is created as :

`<datatype> <variable_name>`

On the similar lines, an object variable can be created as :

`<class_name> <object_name>`

e.g., `Point point1`

`Rectangle rect1`

Declarations do not create new objects. The statement `Point point1` does not create a new Point type object ; it just declares a variable, named *point1*, that will be used to refer to a Point type object. The reference is empty until assigned, as illustrated in Fig. 7.1. An empty reference is known as a *null reference*.

To create an object you need to instantiate it with the new operator.

2. Instantiating an Object

The **new** operator instantiates a class by allocating memory for a new object. The **new** operator requires *a call to a constructor*. The name of the constructor¹ provides the name of the class to instantiate. The constructor initializes the new object.

Objects receive their storage from the **heap memory** [see Fig. 7.2], which is simply a memory pool area managed by Java Interpreter to create objects dynamically (*i.e.*, during the program execution). The **new** operator returns a reference to the object it created. Often, this reference is assigned to a variable of the appropriate type. If the reference is not assigned to a variable, the object is unreachable after the statement in which the **new** operator appears finishes executing.

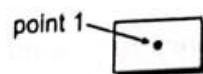


Figure 7.1 Objects as reference.

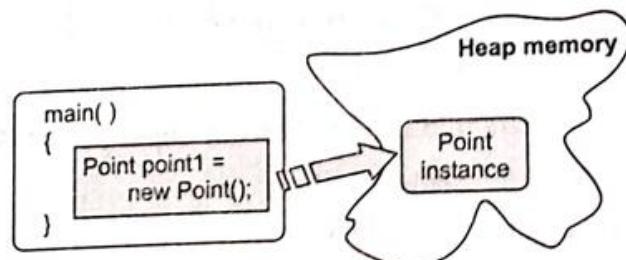


Figure 7.2 Objects get their storage from heap.

3. Initializing an Object

To initialize an object, initial values are passed as constructor's argument.

Consider the code for the Point class :

```
public class Point {
    public int x = 0;
    public int y = 0;
    // A constructor!
```

1. Recall that a constructor is a method having same name as that of the class.

```

public Point(int x, int y) {
    this.x = x;
    this.y = y;
}
}

```

This class contains a single constructor. You can recognize a constructor because it has the same name as the class and has no return type. The constructor in the **Point** class takes two integer arguments, as declared by the code (int x, int y). The following statement provides 20 and 40 as values for those arguments :

```
Point point1 = new Point(20, 40);
```

The effect of the above statement is shown in Fig. 7.3.

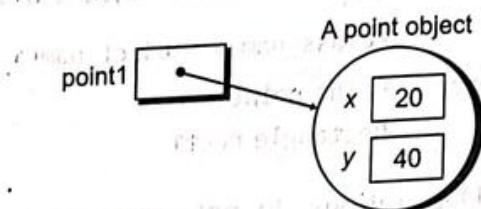


Figure 7.3 An instantiated and initialized object.

7.3.1 Using Objects

Once you've created an object, you probably want to use it for something. You may need information from it, want to change its state, or have it perform some action.

Objects give you *two* ways to do these things :

1. Manipulate or inspect its variables.
2. Call its methods.

7.3.1A Referencing an Object's Variables

The following is the general form of a *qualified name*, which is also known as a *long name*:

objectReference.variableName

e.g., point1.x

refers to data member x of object **point1**. Similarly,

rect1.width

rect1.length

refer to width and length data members of object **rect1** respectively. Following two statements are printing the values stored in two data members of object **rect1**.

```

System.out.println("Rectangle Length :" + rect1.length);
System.out.println("Rectangle Width :" + rect1.width);

```

The first part of the variable's qualified name, *objectReference*, must be a reference to an object. You can use the name of a reference variable here, as in the previous examples, or you can use any expression that returns an object reference variable here, as in the previous examples, or you can use any expression that returns an object reference. Recall that the *new* operator returns a reference to an object.

Note The dot operator is used to refer to members of an object as in **objectreference.membername**.

So you could use the value returned from new to access a new object's variables :

Creates a Rectangle object



refers to the height of the object on the left of dot(.) operation.

```
int height = new Rectangle( ) . height ;
```

This statement creates a new Rectangle object and immediately gets its *height*. In essence, the statement calculates the default height of a Rectangle. Note that after this statement has been executed, the program no longer has a reference to the created *Rectangle*, because the program never stored the reference in a variable.²

However, one thing that you must know is that all member methods of a class can refer to all data members without using the object name e.g.,

```
class Demo {
    int a, b ;
    public setValues( ) {
        a = 5 ;           // see data members a, b are being
        b = 10 ;          // used as a and b and not
    }                   // as objectname.a or objectname.b
    :
}
```

You need to use object name, if you need to access the data member outside the class. However, not all data members can be accessed through *objects* as it depends a lot on their access specifier. This feature will become clear to you after the *Access Specifiers'* discussion later in the chapter.

7.3.1B Calling on Object's Method

You can also use qualified names to call an object's method. To form the qualified name of a method, you append the method name to an object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses i.e., as :

```
objectReference.methodName(argumentList) ;      // while passing arguments
or
objectReference.methodName( ) ;                  // in case of no arguments being passed
```

Suppose the *Rectangle* class has two methods : *area* to compute the rectangle's area and *move* to change the rectangle's origin. Here's how you can call these two methods in two different ways :

```
System.out.println("Area of rect1 :" + rect1.area( )) ;
```

```
....  
rect2 . move(40, 72) ;
```

object name *method name*



method invoked by object on the left of dot(.) operation

The first statement calls *rect1*'s *area* method and displays the results. The second line calls *rect2*'s *move* method.

2. Such an instance is called *temporary instance*. To know more about this concept, you need to refer to Section 7.14 of Chapter 7.

As with instance variables, *objectReference* must be a reference to an object, which can either be an object in itself or an object returned by a statement or function-call e.g.,

```
int areaOfRectangle = new Rectangle(100, 50).area();
```



New object is first created and which then invokes the area() method.

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, the object that *area* is invoked on is the *Rectangle* returned by the constructor.

7.3.2 Controlling Access to Members of a Class – Access Specifiers

Accessing a member of a class depends a lot on *access levels* or *access specifiers*. Access specifiers control access to members of a class from within a Java program. The *access levels* or *access specifiers* supported by Java are : **private**, **protected**, **public** and **default**. Depending upon the access level of a member of-a-class, access to it is denied or allowed. Let us learn more about these access specifiers.

- ◆ **private** access specifier denotes a variable or method as being *private* to the class and may not be accessed outside the class. Accessing will be done through calling one of the *public class methods*.
- ◆ **protected** specifier denotes a variable or method as being *public* to subclasses³ of this class but *private* to all other classes outside the current package. So, derived classes have the ability to access protected variables of its parent.
- ◆ All classes within the same package⁴ have access to protected variables regardless of whether they are subclasses.
- ◆ **public** specifier denotes a variable or method as being directly accessible from all other classes.

If no access specifier is used, then the class member has *default* access – known as *friendly* or *package* access. A package is a logical group of related classes. (Packages are covered in details in chapter 14). Members with package access are available to all classes in the same package, but are not available to any classes in other packages, even subclasses. (You can consider all classes in the same package as friends).

The following chart shows the access level permitted by each specifier :

<i>Specifier</i>	<i>class</i>	<i>subclass</i>	<i>package</i>	<i>world i.e., everywhere</i>
private	✓			
protected	✓	✓	✓	
public	✓	✓	✓	
package	✓		✓	✓

[✓ denotes accessibility].

3. In OOP languages, a class can inherit properties from another class. This is called *inheritance*. The inheriting class is *subclass* and *derived class* and the class being inherited is *super class* or *base class* e.g., we human beings are subclass of *mammals*, which is our super class as we inherit some properties from mammals.
4. A package is a group of logically related classes.

Let us now discuss each access specifier with examples.

Access Specifier private

This is the most restrictive access level. This access specifies should be used to declare members that should only be used by the class. To declare private member, you need to use the keyword **private**. Now consider the following example code :

```
class Alpha {
    private int iamprivate; // private data member
    private void privateMethod() { // private method
        System.out.println("I am privateMethod");
    }
}
```

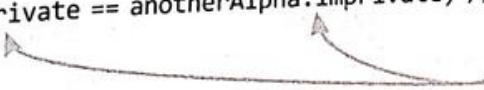
Objects of *Alpha* can modify the *iamprivate* variable and invoke *privateMethod()*. But, objects of other types cannot do so. For example, the *Beta* class cannot.

```
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        // Trying to access private members and methods
        a.imprivate = 10; // Illegal - a compilation error
        a.privateMethod(); // Illegal - a compilation error
    }
}
```

 Accessing private variables or private method outside their class raises error.

But, an object of *Alpha* can access another object of *Alpha*. For example,

```
class Alpha {
    private int imprivate;
    boolean isEqual(Alpha anotherAlpha) {
        if(imprivate == anotherAlpha.imprivate) //legal
            return true;
        else
            return false;
    }
}
```

 See, it's legal to access *imprivate* variable inside the member methods of the same class.

Access Specifier protected

This allows the class itself, subclasses and all classes in the same package have to access the members. Let us consider *Alpha* class is defined inside a Greek package. Now consider following code :

```
package Greek;
class Alpha {
```

```


protected int iamprotected ;           // protected data member
protected void protectedMethod( )      // protected method
{
    System.out.println("I am protected Method");
}
}

```

Let us say another class *Gamma* is also a class in the *Greek* package :

```

package Greek ;
class Gamma {
    void accessMethod( )  {
        Alpha a = new Alpha( );
        a.iamprotected = 10 ;           // legal
        a.protectedMethod( ) ;        //legal
    }
}

protected members can be accessed from within classes
belonging to the same package to which the class of
protected members belongs.

```

Let us have another class *Delta* that is derived from *Alpha* but lives in a different package - *Latin*. *Delta* class cannot access *iamprotected* and *protectedMethod* on objects of type *Alpha*, but access with objects of type *Delta*.

```

package Latin ;
import Greek.*;
class Delta extends Alpha {
    // it means class Delta is inheriting members of class Alpha.
    void accessMethod(Alpha a, Delta d) {
        a.iamprotected = 10 ;           // Illegal
        a.protectedMethod( ) ;         //Illegal
        d.iamprotected = 10 ;           //legal
        d.protectedMethod( ) ;         //legal
    }
}

```

Access Specifier **public**

Any class, in any package has access to a class's *public* members. For example,

```


package Greek ;
class Alpha {
    public int impublic ;
    public void publicMethod( ) {           // public data member
        System.out.println("I am public Method");
    }
}

// public method

```

Let us say Beta is defined in another class.

```
package Roman ;
import Greek.* ;
class Beta {
    void accessMethod( ) {
        Alpha a = new Alpha( ) ;
        a.impPublic = 10 ;           // legal
        a.packageMethod( ) ;        //legal
    }
}
```



public members are accessible in all classes.

Access Specifier friendly(Default)

Java assumes this access specifier if you don't explicitly set any member's access level. This assumes that all classes in the same package are trusted friends.

```
package Greek ;
class Alpha {
    int impackage ;
    void packageMethod( ) {
        System.out.println("I am package Method") ;
    }
}
```

Let us say Beta class is also inside the same package Greek.

```
package Greek ;
class Beta {
    void accessMethod( ) {
        Alpha a = new Alpha( ) ;
        a.impackage = 10 ;           // legal
        a.packageMethod( ) ;        //legal
    }
}
```



*friendly access as both the classes
(Alpha & Beta) belong to same package.*

Following Table 7.1 summarizes the accessibility offered by various access specifiers.

Table 7.1 Accessibility of Access specifiers

Access Specifier	Accessible by classes in the same package	Accessible by classes in other packages	Accessible by subclasses in the same package	Accessible by subclasses in other packages
Public	Yes	Yes	Yes	Yes
Protected	Yes	No	Yes	Yes
Package (default)	Yes	No	Yes	No
Private	No	No	No	No

Notes The default access specifier is **friendly**, but it is not a keyword. Friendly means all **friends** (classes in the same package) can access it.

All these access specifiers are applicable to both the **class variables (static variables)** and **instance variable**, and also to **class methods (static methods)** and **other methods**.

Getters and Setters (Mutator) Methods in classes

When writing new classes, issue of access control should be dealt with care. Recall that making a member of a class **public** makes it accessible from anywhere, including from other classes. On the other hand, a **private** member can only be used in the class where it is defined.

Many programmers opine that (almost) all member variables should be declared **private**, as this gives complete control over what can be done with the variable. Even if the variable itself is **private**, you can allow other classes to find out what its value is by providing a public **accessor method** that returns the value of the variable.

For example, if your class contains a **private** member variable, *marks*, of type **double**, you can provide a method as :

*Getter method for
private variable
marks*

```
public double getMarks() {  
    return marks; // returns the value of marks  
}
```

*See, the accessor method has the same
return type as that of the data member
whose value it is returning.*

By convention, the name of an accessor method for a variable is obtained by capitalizing the name of variable and adding "get" in front of the name. So, for the variable *marks*, we get an accessor method named "get" + "Marks", or *getMarks()*. Because of this naming convention, accessor methods are more often referred to as **getter methods**. A getter method provides "*read access*" to a variable.

You might want to make it possible for other classes to specify a new value for the variable (*i.e.*, to allow "*write access*" to a private variable). This is done with a **setter method**. (also called **mutator method**.)

A setter method (or Mutator method) is a method that sets/changes the value of a data-member of the class. Mostly setter methods are provided for private members.

The name of a setter method generally consists of "set" followed by a capitalized copy of the variable's name, and it should have a parameter with the same type as the variable *e.g.*, a setter method for the variable *marks* could be written as :

*Setter method for
private variable
marks*

```
public void setMarks( double newMarks ) {  
    marks = newMarks; // sets/changes the value of marks  
}
```

*See, the parameter of a setter method
has the same return type as that of the
data member whose value it sets.*

It is actually very common to provide both a getter and a setter method for a private member variable. Since this allows other classes both to see and to change the value of the variable.

7.4 Encapsulation

Encapsulation means that a single thing encloses several smaller things that are its parts. This is similar to the process of abstraction. In OO technology, several simple things are encapsulated into a more complex thing that is called the class. The simple things that are encapsulated by the class are called the members of the class.

A class consists of :

- (i) **Data members** that contain information necessary to represent the class.
- (ii) **Methods** that perform operations on the data members of the class.

Internal details of the class should be visible only inside the class using a philosophy called *information hiding*. Why is information hiding necessary ? You know it already, but still, let us revise it.

- ◆ Users should not be concerned with unnecessary details.
- ◆ Users may do harm if they try to modify the internals of a class.

7.5 Visibility Modifiers

In Java we accomplish **encapsulation** through the appropriate use of *visibility modifiers* (a modifier is a Java reserved word that specifies particular characteristics of a method or data value).

An **access specifier** (visibility modifier) defines who (which function or method) is able to use this method. Access specifiers can be :

- ◆ **public** which means that anyone can call this method (public methods are also called service methods ; A method created simply to assist a service method is called a support method).
- ◆ **private** which means that only the methods in the same class are permitted to use this method.
- ◆ **protected** which means that methods in this class and methods in any subclasses may use this method.
- ◆ (nothing) i.e., **friendly** (it is not a keyword) when any classes in this particular package or directory may access this method.

We have already talked about these access specifiers in details. Let us now use these access specifiers for learning more about scope and visibility.

7.6 Scope and Visibility Rules

The scope rules of a language are the rules that decide, in which part(s) of the program a particular piece of code or data item would be known and can be accessed therein.

Here **visibility** is a related term. Visibility refers to whether you can use a variable from a given place in the program.

Java offers following levels of scope and visibility :

- ◆ Data declared at the class level can be used by all methods in that class.
- ◆ Data declared within a method can be used only in that method.
- ◆ Data declared within a method is called *local data*.
- ◆ Variables that are declared in block i.e., *local variables* are available to every method *inside of that block*.
- ◆ Variables declared in interior blocks are not visible outside of that block.
- ◆ Variables declared in exterior blocks are visible to the interior blocks.

The program part(s) in which a particular piece of code or a data value (e.g., variable) can be accessed is known as the piece-of-code's or variable's **Scope**.

For example

```

class sample {
    int x ;
    /* Variables declared here are global to this class i.e. x is globally available
       in class sample. If public, may be accessed outside of the class */

    void method1( ) {
        byte by ;
        /* variables declared here are local to method1. Are not be visible
           outside of method or class i.e., variable by is visible only inside method1
           and not outside it */
    }

    void method2( ) {
        /* variables declared here are local to method2.
           (Note : formal parameters are local variables) */
    }

    void method3( ) {
        int a = 0, b = 1, c = 2 ; // a, b, c are available throughout method3( )
        while(a < b) {
            int temp ; // temp is available only inside the block of while
            temp = a ; // a, b, c are available here also
            a = b ;
            b = temp ;
        }
        .....
        // temp not accessible here but a, b, c are
    }

    public static boolean method4 (int parm1, double parm2) {
        // For this method, parm1 and parm2 will be visible throughout the method.
    }
}

```

Resolving identifiers i.e., local variable hiding global variable

If you have a *global variable* (i.e., variable available to the entire class and its members) with the same name as a *local variable*, (i.e., variable declared inside a method or block) the system resolves the name to the most local **scope** available i.e., *most local variable* with the same name will be considered. This means that a *local variable* having the same name as that of *global variable* hides the *global variable*. To understand this, consider the following program :

```

/* Small test program to show global variable effect */

public class VarHiding {
    // global variable "globalVar" is set to 0
    public static int globalVar = 0 ;
    public static void main(String[ ] args) {
        test(5) ;
        System.out.println(" First test :" + globalVar) ;
        test2 (7) ;
        System.out.println("Second test :" + globalVar) ;
    }
}

```

```

public static void test(int globalVar)
    // test uses the same name as our globalVar variable {
        globalVar = globalVar ;                                Local to method test
                                                                All of these refer to local globalVar

        // Here local variable namely globalVar will be considered
        // Global variable namely globalVar remains hidden here
        System.out.println("Inside of test :" + globalVar) ;

    }

public static void test2(int value)
    // test two uses a different name. To resolve variable globalVar, we must go
    // higher in the "stack" {
        globalVar = value ;                                  Refers to global globalVar

        // Here globalVar variable namely globalVar will be considered.
        System.out.println("Inside of test2 :" + globalVar) ;
    }
}

```

The scope and visibility rules are summarized in following Table 7.2 :

Table 7.2 Scope and Visibility Rules

	<i>Scope</i>	<i>Visibility</i>	<i>Purpose</i>
public classes and variables	public	visible to all classes	Methods and variables of interest to users of the class.
protected classes and variables	protected	visible to classes outside the package that inherit the class, also to all classes in the package.	Methods and variables of interests to third parties who may extend your class.
classes and variables declared with nothing	default aka package aka friendly	visible to all classes of the package.	Methods and variables involved in cross-class communication within the package.
private classes and variables	private	visible only within the class, not by inheritors, not by other classes in the package.	Variables and methods that should not or would not be changed by someone extending the class. Proper functioning of the class depends on them working precisely as written.
Variables declared inside method bodies or other blocks	local	visible only to the block of the method in which the variables were declared.	Temporary working variables.

The output produced is
 Inside of test : 5
 First test : 0
 Inside of test2 : 7
 Second test : 7

Similar scope rules are also applicable to classes as well as summarized below :

	<i>Same class</i>	<i>Other class same package</i>	<i>Subclass other package</i>	<i>Any class</i>
<i>public class</i>				
public member	Yes	Yes	Yes	Yes
protected member	Yes	Yes	Yes	No
member	Yes	Yes	No	No
private member	Yes	No	No	No
<i>protected class</i>				
public member	Yes	Yes	No	No
protected member	Yes	Yes	No	No
member	Yes	Yes	No	No
private member	Yes	No	No	No
<i>private class (an inner class) Class defined inside another class or method</i>				
public member	Yes	No	No	No
protected member	Yes	No	No	No
member	Yes	No	No	No
private member	Yes	No	No	No

Let Us Revise

- ❖ Composite datatypes are based on fundamental or primitive datatypes.
- ❖ Composite datatypes are also called user defined datatype.
- ❖ A user defined datatype is defined through a class but it does not contain main method.
- ❖ Objects are created through new operator.
- ❖ Objects are allocated memory from heap, the free storage area for dynamic memory allocation.
- ❖ The public members of objects are accessed through dot operator.
- ❖ The private members are accessible only inside their own class.
- ❖ The protected members are accessible inside their own class, subclass and package.
- ❖ The public members are accessible everywhere in the program.
- ❖ The default (friendly or package) members are accessible inside their own class as well to classes in the same package.
- ❖ Encapsulation means that a single thing encloses several smaller things that are its parts.
- ❖ Java accomplishes encapsulation through appropriate use of visibility modifiers (public, private, protected or default).
- ❖ Scope refers to the region within which a variable is accessible.
- ❖ A local variable having the same name as that of global variable (having class scope), hides the global variable.

Solved Problems

1. Which of the following declarations are illegal and why?

- (i) class Parser { ... }
- (ii) public class EightDimensionalComplex { ... }
- (iii) private int i ;
- (iv) private class Horse { ... }
- (v) public protected int x ;
- (vi) default Button getBtn() { ... }

Solution. Declarations (v) and (vi) are illegal. The reasons are :

- Declaration (v) is illegal because there can be at most one access specifier. Here two access specifiers (public protected) are simultaneously used.
- Declaration (vi) is illegal because default is used here to specify the access specifier but default is not a keyword and it cannot be used as an access specifier. To specify default access level, no access specifier is specified.

2. Out of declarations given in problem 1, which one is having default access?

Solution. Declaration (i).

3. Compare a class as a user defined datatype and a class as an application.

Solution. In Java, all functionality is enclosed in classes. But in order for a class to be user-defined datatype, it should act different from that of an application. That is, it should not include main method in it. Although you can create instances of classes containing main method, they should not be referred to as user-defined datatype. Such classes (containing main method) are more analogous to application than a datatype as a datatype's functionality is implemented in another class not in its own class.

4. Design and implement a public class namely MusicStore as with one public method, displayHoursOfOperation. This method should display the daily hours of operation to the standard output device. The expected output is as follows :

Store Hours : Daily : 9:00 AM - 9:00 PM

Solution.

```
public class MusicStore {
    void displayHoursOfOperation( ) {
        System.out.print("Store Hours :");
        System.out.println("Daily : 9:00 AM - 9:00 PM");
    }
}
```

5. Design and implement DemoMusicStore as a public class with method main. Method main should perform the following tasks :

- Creates an instance of MusicStore.
- Invokes the displayHoursOfOperation method to display the daily hours of operation to standard output.
- Terminates the program with exit function.
(Use System.exit(0) to exit from the program i.e., to terminate the program. Usually value 0 is passed to depict normal termination. You can pass any other value to suggest abnormal termination of the program.)

```

Solution. public class DemoMusicStore {
    public static void main(String[ ] args) {
        MusicStore ms = new MusicStore( );
        ms.displayHoursOfOperation( );
        System.exit(0);
    }
}

```

6. Now list the complete program with both the classes you have created.

```

Solution. public class MusicStore {
    void displayHoursOfOperation( ) {
        System.out.print("Store Hours :");
        System.out.println("Daily : 9:00 AM - 21:00 PM");
    }
}

public class DemoMusicStore {
    public static void main(String[ ] args) {
        MusicStore ms = new MusicStore( );
        ms.displayHoursOfOperation( );
        System.exit(0);
    }
}

```

7. Out of the two classes you created, identify the user-defined type and the application.

Solution. Out of above given two classes, the class MusicStore is the user-defined type, for :

- It does not contain method main
- Its functionality is implemented through another class.

The class DemoMusicStore is the application as it contains the method main wherefrom all execution takes place.

8. Write a class Convert with methods as follows :

- (a) takes 4 arguments representing miles, yards, feet and inches and converts them into kilometers,
- (b) takes an argument representing degrees Fahrenheit and converts it to degrees centigrade.
- (c) a kilobyte is interpreted in two ways : sometimes it is 1000 bytes (actually correct), but often (and traditionally) it is 2^{10} which is 1024. Similar discrepancies arise for mega, giga, tera and peta (each is 1000 (or 2^{10}) times the previous one). The function should take the 10^3 (standard kilo) and give the equivalent value using 2^{10} as a kilo for all the above.

Solution. class Convert{

```

    void distance(int miles, int yards, int feet, int inches) {
        long mkm, mm, mc, ykm, ym, yc, fkm, fm, fc, ikm, im, ic, nkm, nm, nc;
        mkm = (int)(miles * 1.609);
        mm = (int)(miles * 1609 - mkm*1000);
        mc = (int)(miles * 160900 - (mkm*1000*100+mm*100));
        ykm = (int)(yards * 0.0009144);
        ym = (int)(yards * 0.9144 - ykm*1000);
        yc = (int)(yards * 91.44 - (ym*1000*100+ykm*100));
        fkm = (int)(feet * 0.0003048);
        fm = (int)(feet * 0.3048 - fkm*1000);
    }
}

```

```

        fc = (int)(feet * 30.48 - (fkm*1000*100+fm*100)) ;
        ikm = (int)(inches * 0.0002540) ;
        im = (int)(inches * 0.02540 - ikm*1000) ;
        ic = (int)(inches * 2.540 - (ikm*1000*100+im*100)) ;
        nkm = mkm+ykm+fkm+ikm ;
        nm = mm+ym+fm+im ;
        nc = mc+yc+fc+ic ;
        System.out.println("Equivalent distance in Kilometer,meter, centimeter is : ") ;
        System.out.println(nkm+" KM "+nm+" M "+nc+" CM ") ;
    }

    void temperature(float fahrenheit) {
        float centigrade ;
        centigrade = (fahrenheit - 32)/1.8F ;
        System.out.println("Temperature in centigrade is : "+centigrade) ;
    }

    void memoryunit(long skilo) {
        long giga, mega, kilo ;
        giga = skilo / (1024*1024*1024) ;
        skilo = skilo % (1024*1024*1024) ;
        mega = skilo / (1024*1024) ;
        skilo = skilo % (1024*1024) ;
        kilo = skilo / (1024) ;
        skilo = skilo % (1024) ;
        System.out.println("Equivalent value in standard units : ") ;
        System.out.println(giga + "GB"+ mega + "MB"+ kilo +"KB") ;
    }
}

```

9. Class can be used to define user-defined datatype. How ?

Solution. A class can contain data elements as well as the definitions of operations that can be performed on the class itself. Further, one can use a class as a data type to declare variables. Thus a class can well said to be a user defined data type. For instance, let us consider a type specification for a class Book :

```

class Book {
    String Title, Author, Subject ;
    int Edition ;
    float Price ;
    public Book( ) {
        this.Title= " " ;
        this.Author= " " ;
        this.Subject= " " ;
        this.Edition= 0 ;
        this.Price= 0 ;
    }
    public Book (String Title, String Author, String Subject, int Edition, float Price) {
        this.Title= Title ;
        this.Author= Author ;
        this.Subject= Subject ;
    }
}

```

```

        this.Edition= Edition ;
        this.Price= Price ;
    }

    public void display() {
        System.out.println ("Title : " + Title) ;
        System.out.println ("Author : " + Author) ;
        System.out.println ("Subject : " + Subject) ;
        System.out.println ("Edition : " + Edition) ;
        System.out.println ("Price : " + Price) ;
    }
}

```

Now in order to create objects / variables of this data type, one can use either of the following statements:

```

Book b1( ) ;
Book b2 ("Core Java", "Gary Cornell", "Java", "1", 500) ;

```

The first statement creates an object of class book whose members are set to blank, whereas the second statement creates a variable whose members are set according to the parameters passed.

10. What will be the output of following code fragment ?

```

class Test {
    public static int varOne = 10 ;
    public static void method1( ) {
        int varOne = 25 ;
        System.out.println(varOne) ;
    }

    public static void method2( ) {
        System.out.println(varOne) ;
    }

    public static void main( String[ ] args) {
        method1( ) ;
        method2( ) ;
    }
}

```

Solution. 25

10

The reason behind it is that in method1(), local variable *varOne* hides the global *varOne*, hence value of local *varOne* which is 25 gets printed.

In method2(), no variable is hiding global *varOne*, hence its value 10 gets printed.

11. Can a static class member of public class get accessed by

- (i) same class
- (ii) other class in the same package
- (iii) class in any other package ?

Solution. (i) Yes, (ii) Yes, (iii) No.

12. Can a public member of public class get accessed by
 (i) same class (ii) other class in the same package (iii) class in any other package ?

Solution. (i) Yes, (ii) Yes, (iii) Yes.

13. Can a public member of private class get accessed by

- (i) same class (ii) other class in the same package (iii) class in any other package ?

Solution. (i) Yes, (ii) No, (iii) No.

Glossary

Access specifier Way to control access of class members.

Composite Datatype Datatype based on primitive datatypes.

User defined Datatype Datatype created by the user.

Scope Region within which a variable/piece-of-code is accessible.

Local Variable Variable declared inside a method or block.

Global Variable Class variable which is available to the entire class.

Assignments

TYPE A : VERY SHORT ANSWER QUESTIONS

- What is datatype ?
- What is composite datatype ? What is user defined datatype ?
- Can you refer to a class as a composite type / user-defined type ?
- Can every class be referred to as a user-defined datatype ?
- Which keyword can protect a class in a package from accessibility by the classes outside the package ?
 (a) private (b) protected
 (c) final (d) don't use any keyword at all (make it default)
- We would like to make a member of a class visible in all subclasses regardless of what package they are in. Which one of the following keywords would achieve this ?
 (a) private (b) protected (c) public (d) private protected
- The use of protected keyword to a member in a class will restrict its visibility as follows :
 (a) visible only in the class and its subclass in the same package
 (b) visible only inside the same package
 (c) visible in all classes in the same package and subclasses in other packages
 (d) visible only in the class where it is declared
- Which of the following keywords are used to control access to a class member ?
 (a) default (b) abstract (c) protected (d) interface (e) public
- The default access specifier of class members is _____ ?
- What is encapsulation ? What does a class encapsulate ?
- What will be the scope of :
 (a) a public class ? (b) a protected class ?
 (c) a default class ? (d) a private class ?

TYPE B : SHORT ANSWER QUESTIONS

1. What is friendly access of class members ?
 2. What is public access of class members ?
 3. How are private members different from public members of a class ?
 4. How are protected members different from public and private members of a class ?
 5. How does a class enforce information hiding ?
 6. Name various visibility modifiers in a class. Is visibility related to scope somehow ?
 7. Define scope and visibility. Give an example to explain scope and visibility.
 8. Is protected access somewhat similar to default (friendly) access ? Explain with the help of an example.

TYPE C : LONG ANSWER QUESTIONS

1. Class can be used to define user-defined datatype. How ?
 2. Why can't every class be termed as user-defined datatype ? How is a user-defined datatype different from an application ?
 3. Where can the following members of a class be accessed ?
(i) private (ii) protected (iii) public (iv) members with default access.
 4. Define a class for *Dates*. Add required functions e.g., constructors, printing a date, validating a date etc.
 5. Use the above defined *Date* datatype in a different class for creating objects such as birthDate, joinDate, examDate etc.
Validate these dates. Try verifying them by passing invalid date e.g., 33/13/03.
 6. Out of classes defined in questions 4 and 5, which one is a user-defined datatype and which one is an application ? Justify your answer.
 7. How does Java resolve variables having same name ? Give code example.
 8. Implement a class to represent 2D point (x, y). You should be able to print the coordinates of the point and also compare the two points for equality.
 9. Write a program to implement a circle class. You should be able to display the radius, area and perimeter of the circle.
 10. Write a program to implement a student class. Have a result method as member of the class. This method should takes marks in three subjects as parameters and display result of the student.

CHAPTER

8

Functions (Methods)

8.1 Introduction

A program is a set of instructions given to computer. These instructions initiate some action and hence sometimes called executable instructions. In Java programs, the executable instructions are specified through **methods** or **functions**. And as you know that a *method* or a *function* is a sequence of some declaration statements and executable statements. In other programs, *methods* are known as *functions*, sometimes *procedures*, *subprograms* or *subroutines*.

In Java, which is strictly object-oriented, any action can take place through *methods* and methods have to exist as a part of a class. Programmers design object-oriented programs by deciding first what specific actions have to be performed and what kind of objects should perform them. This chapter is dedicated to the discussion of functions that you write to define methods of classes. Let us begin with how you can define functions in Java. But, hey, Wait ! First of all shouldn't we know why do we use functions at all ?

In This Chapter

- 8.1 Introduction
- 8.2 Why Functions ?
- 8.3 Function/Method Definition
- 8.4 Accessing a Function
- 8.5 Pass by Value (Call by Value)
- 8.6 Call by Reference
- 8.7 Returning from a Function
- 8.8 Pure and Impure Functions
- 8.9 Function Overloading
- 8.10 Calling Overloaded Functions
- 8.11 Constructors
- 8.12 Types of Constructors
- 8.13 The this Keyword
- 8.14 Temporary Instances
- 8.15 Constructor Overloading
- 8.16 Recursive Functions
- 8.17 Recursion in Java

8.2 Why Functions ?

There are at least three reasons why we use methods :

- (i) to allow us to cope with complex problems,
- (ii) to hide low-level details that otherwise obscure and confuse, and
- (iii) to reuse portions of code.

Let's examine each of these points in details.

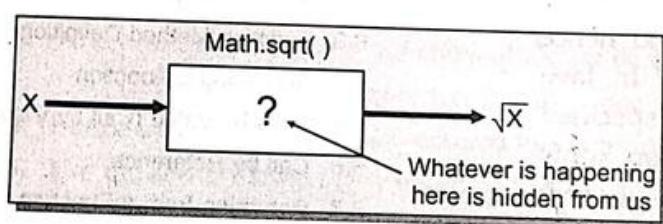
(i) **To cope with Complexity.** When programs become more and more complex i.e., when they gain in size, generally, sooner or later, they become unruly. One of the most powerful techniques to cope with complexity in computer programs is affectionately known as "divide and conquer" — that is, to take a complex task and divide it into smaller, more easily understood tasks. In Java, those "smaller, more easily understood tasks" are implemented as *methods*. Once a method is defined, it is "put aside" and used. The task of the method is thereafter "solved", and we needn't concern ourselves with the details any longer.

(ii) **Hiding Details.** Another important use of methods is to create "black boxes". Once a method is defined, it can be used in a program. At the level of using it, we needn't concern ourselves with how the method's task is performed. We just get the task done through methods. Basically, when we use a method we are *delegating* the task of solving a low-level

problem to a method. To solve it ourselves could become distracting and unnecessarily time-consuming. We are, in a sense, treating the method as a black box, because we accept the result without concern for the details. This is illustrated in Fig. 8.1.

The question mark in Fig. 8.1 emphasizes that the details of calculating a square root are hidden from us.

Figure 8.1 A method as a black box.



(iii) **Reuse.** Once a task is packaged in a method, that method is available to be accessed, or "called", from anywhere in a program. The method can be reused. It can be called more than once in a program, and it can be called from other programs. This is illustrated in Fig. 8.2. Reuse is an important concept in object-oriented programming languages like Java. The practice is sometimes called, "Write once, use many". Rather than "reinventing the wheel", we build on previous efforts or on the efforts of other programmers — we reuse what precedes us.

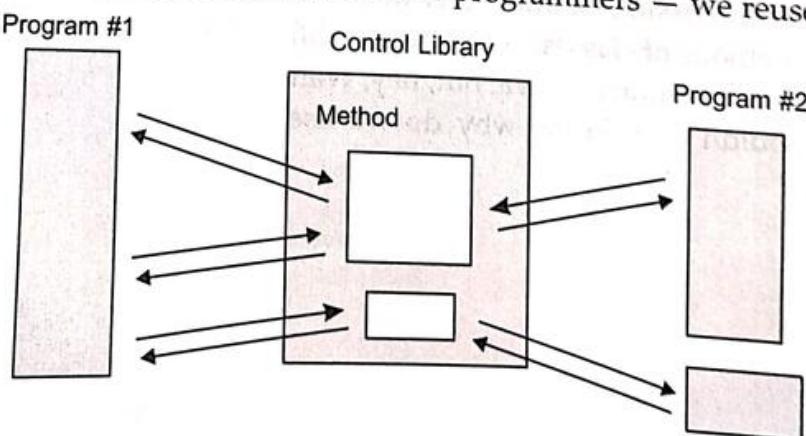


Figure 8.2 Method reuse.

After learning about "why functions", let us now learn to write simple functions in Java. In the next section we'll learn how to define simple methods in the Java language.

8.3 Function/Method Definition

In Java, a function (or method) must be defined before it is used anywhere in the program. The general form of a function / method definition is as given below :

```
[access-specifier] [modifier] return-type function-name (parameter List) {
    body of the function
}
```

where

access specifier can be either **public** or **protected** or **private**. These keywords are used to determine the type of access to the function. Default is *friendly*, which is not a keyword. We shall talk about access specifiers in details a little later in the chapter.

modifier can be one of : **final**, **native**, **synchronized**, **transient**, **volatile**. A **final** method means that the functionality defined inside this very method can never be changed. Discussion of other modifiers is beyond the scope of this book.

return_type specifies the type of value that the return statement of the function returns. It may be of any valid Java data type. If no value is being returned, it should be **void**.

Function-name It has to be valid Java identifier. The conventions generally followed for method-naming are :

- (i) should be meaningful.
- (ii) it should begin with a *lowercase letter*. For names having multiple words, join the words and begin each word with uppercase letter e.g.,

printReportCard
getMarks

- (iii) The method names should, generally, begin with a verb followed by one or more nouns e.g.,

readData
findFile
calculateInterestAmount

Parameter list The *parameter list* is a comma-separated list of variables of a function referred to as its *arguments* or *parameters*. A function may be without any parameters, in which case, the *parameter list* is empty.

Here are some examples of functions :

```
(a) int absval( int a ) {
    
    Parameter list has only one parameter
    return (a < 0 ? - a : a);
    // this function is returning absolute value of variable a
}
```

(b) `public static int maximum(int a, int b)`

```
{     int ans ;
      ans = ( a > b ) ? a : b ;
      return ans ;
}
```



Parameter list has two parameters

(c) `public static void threestars ()`

```
{
    System.out.println(" * * *");
}
```



Parameter list is empty

Note While naming a function or method, just make sure that it should be a legal identifier and should be meaningful.

From the above examples, it is clear now that the parameter declaration list for a function takes this general form :

... `function_name(type varname1, type varname2, type varnamen)`

The conventions generally followed for method-naming are :

(i) should be meaningful.

(ii) it should begin with a lowercase letter. For names having multiple words, join the words and begin each word with uppercase letter e.g.,

`printReportCard`
`getMarks`

(iii) The method names should, generally, begin with a verb followed by one or more nouns e.g.,

`readData`
`findFile`
`calculateInterestAmount`

Before we start writing our own methods, let us have a look at some predefined methods that we can use in your programs, such as mathematical functions available through `Math` class.

For instance, to obtain square root of a number say `n`, we may write

```
int n = 25, s ;
s = Math.sqrt(n) ;
```

Let us examine the last statement which invokes the mathematical function :

```
s = Math.sqrt(n) ;
```

The value returned by function is stored inside this variables. Similarly, if we have a statement such as

```
double b = Math.cos(60) ;
```

we easily can identify that :

the name of method/function is `cos()`

the argument being passed to it is `60`

the returned value will be stored in variable `b` of double type.

Can you make out what is the *return type* (the datatype of the value being returned) ?

Well, generally we receive the returned value in a variable of same datatype. (Although, it is not necessary, you may even decide to receive the returned value in a different but compatible datatype.)

Methods live in classes

The methods live inside classes. In order to exist within a Java program, a method has to exist inside a class. A Java program can have many classes and each class can have several methods. And one class in every program contains a `main()` method. The `main()` method is very important as it tells the program where to start. The general rule is that there should be one `main()` method in a program i.e., inside one class only, in a program. Although, this rule can be broken, but it is an advanced topic and not recommended.

8.3.1 Function Prototype and Signature

Once again, have a look at examples of functions (a) and (b) given in Section 8.3. The first line of the function definition is the prototype of the function i.e., the prototypes of functions (a) and (b) defined above are :

`int absval(int a)`

and `public static int maximum(int a, int b)`

A function prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.

With the help of a function prototype, a compiler can carefully compare each use of function with the prototype to determine whether the function is invoked properly i.e., the number and types of arguments are compared and any wrong number or type of the argument is reported. This helps the compiler pinpoint errors easily.

A **Function Prototype** is the first line of the function definition that tells the program about the type of the value returned by the function and the number and type of arguments.

Function Signature

A related term here is *function signature*. A function signature basically refers to the *number and types of arguments*. It is a part of the function prototype. Function signature and return type make a function's prototype. But sometimes the term *function signature* is used to refer to a function prototype.

Let us use function (b) defined earlier through following program in order to understand proper usage of functions.

```

1. public class DemoMethod
2. {
3.     public void DemoMethodTest( )
4.     {
5.         int x = 85 ;

```

The type of value being returned from the function. Since this function is not returning any value, its return type is `void`, which means this function returns no value.

```

6.     int y = 26 ;
7.     int z = DemoMethod.maximum(x, y) ;
8.     System.out.println("The larger one is" + z) ;
9. }      // main ends here

```

maximum() function of DemoMethod class is invoked here

```

10.    public static int maximum( int a, int b ) {
11.        if(a > b)
12.            return a ;
13.        else
14.            return b ;
15.    }      // function maximum ends here
16. }      // class definition ends here

```

This keyword (static) means that this very method is now a class method; it will be called through class name rather than through object. See, in above function, this has been invoked through its class DemoMethod

The type of value being returned from the function maximum()

Parameters being received by the function maximum(). See, it is receiving two integers namely a and b.

The above program will produce result as :

The larger one is 85

Let us now walk through the definition of function *maximum()*.

The reserved word **public** is a *modifier*; it specifies the *visibility* attribute of the method. Most Java methods are *public*, meaning they are accessible from other methods, perhaps from methods in other classes. Visibility attributes are discussed in more details later.

The reserved word **static** is also a modifier; it identifies the method as a *class method* (sometimes called a *static method*). It means that the method is not called through an instance of a class but, rather, through the class itself. Understanding this is tricky, so let's review the syntax for calling a class method vs. calling an instance method. (*This was discussed earlier in previous chapters*). The method *sqrt()* of the *Math* class is a *class method*.

It might be called as follows :

```

double x = 25.0
double y = Math.sqrt(x) ;           // sqrt( ) is a 'class method'

```

sqrt() is a 'class method' as it is being invoked through class name.

// sqrt() is a 'class method'

The method *substring()* of the *String* class is an "instance method". It might be called as follows :

```

String s1 = "hello" ;
String s2 = s1.substring(0, 1) ;           // substring( ) is an 'instance method'

```

s1 is an object; the function being invoked through it (substring()) is instance method.

// substring() is an 'instance method'

Do you see the difference? Preceding `sqrt()` is "Math." which identifies the class in which `sqrt()` is defined. Preceding `substring()` is "s1." which is an instance variable of the `String` class. "Math" is the name of a class (hence "class method"), whereas "s1" is the name of an instance variable (hence "instance method").

The preceding discussion is put in terms of the `maximum()` method as follows:

the maximum() method is a static method, or a class method, of the DemoMethod class.

Returning to the signature for the `maximum()` method, the next word is the reserved word `int`. This identifies the return type of the method. The `maximum()` method returns a value of type `int`. Anywhere an integer value can be used — for example, in an expression — the `maximum()` method can be used. All Java methods must include a return type in their definition. In some cases, a method has nothing to return, (e.g., `println()`), and is defined to return a `void`.

Next comes the name of the method followed by a set of parentheses. The name of the method confirms to the naming conventions for Java identifiers given earlier. Within the parentheses are the arguments passed to the method. In the case of `maximum()`, it receives two arguments of type `int`. The names of the arguments are arbitrary. They are used in the definition of the method (see lines 12-15), but they have no relationship to the names of variables in the calling program.

Immediately following the signature is the definition of the method within a set of braces `{ }`. The `maximum()` method is defined in lines 11-16. The operation is pretty simple, so we needn't dwell on the details. The reserved word `return` causes an immediate exit from the method, with control returning to the calling method. The statement `return` is followed by an *expression, a variable name, or the value to be returned to the calling program*. A compile error occurs if the type of the value returned does not match the type appearing the signature (notwithstanding automatic promotion, for example, of `int` to `double`).

Use of void

As you know about `void` data type that it specifies an empty set of values and it is used as the return type for functions that do not return a value. Thus, a function that does not return a value is declared as follows:

```
void function-name (parameter list) ;
```

By declaring a function's return type `void`, one makes sure that the function cannot be used in an assignment statement. The functions returning some value can be used in expressions and assignment statements.

8.4 Accessing a Function

A function is *called* (or *invoked*, or *executed*) by providing the function name, followed by the parameters being sent enclosed in parentheses. For instance, to invoke a function whose prototype looks like

```
float area (float x, float y)
```

the function call statement may look like as shown below:

```
z = area (x, y) ;
```

where x, y, z have to be **float** variables. The syntax of the function call is very similar to that of the declaration, except that the *type* specifiers are missing. Whenever a function call statement is encountered, the control (program control) is transferred to the function, the statements in the function-body are executed, and then the control returns to the statement following the function call. Following program (8.1) uses function prototyping, function definition and function calling.

Program 8.1

Program to print cube of a given number using a function.

```
public class prg13_1 {
    static float cube (float a) { .....  
        float n = a * a * a ;  
        return n ;  
    }  
    public static void main(String args[ ]) {  
        float x = 7.5f, y = 0 ;  
        y = cube (x) ; .....  
        System.out.println ("The cube of " + x + " is " + y) ;  
    }  
}
```



Control gets transferred to
function definition

//function call statement

The output produced is

The cube of 7.5 is 421.875

Dotted Lines show the transfer of control.

The above program first declares a function **cube** by providing its prototype. Then after accepting the number it invokes the function **cube** and assigns the return value of it to the variable **y**. As soon as the function call statement is encountered, the control gets transferred to function-body and all its statements are executed. With the execution of a return statement, the control returns to the statement immediately following the function call statement.

When a function, that expects some arguments, is invoked, its call statement must provide the argument values being sent and the number and type of argument values must be the same as defined in the function prototype. Also the order of argument values must be the same as defined in the function prototype. For instance, if a function **volume** expects two values : first of **float** type and the second as **int** type then its call statement must look like as :

volume (*a, b*) ;

where *a* must be a **float** value and *b* must be an **int** value.

A function, that does not expect any argument, is invoked by specifying empty parentheses. For instance, if a function **message** does not expect any argument, it will be invoked as :

message () ;

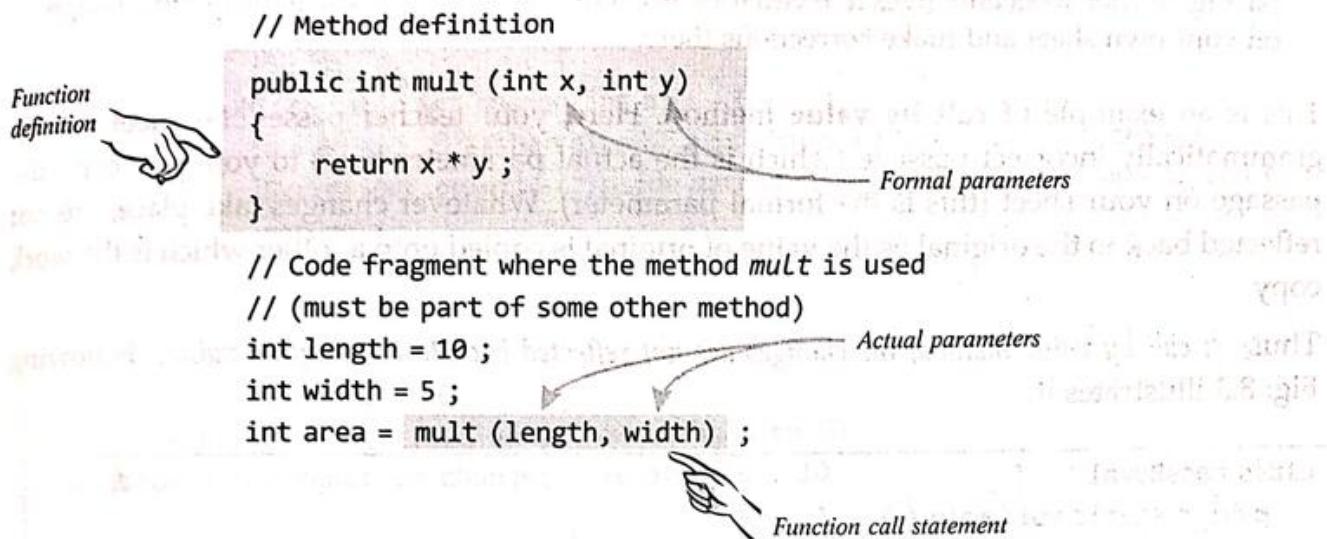
Also, some functions are used in expressions, but only those functions may be used in expressions that return a value.

TIP

Only the functions returning a value can be used in expressions.

8.4.1 Actual and Formal Parameters

After the discussion we have had so far, you have seen that there are parameters in the function (or method) definition and in the statement that invokes the function i.e., the function call statement. The parameters that appear in function definition are called *formal parameters* and the parameters that appear in function call statement are called *actual parameters*. Consider the following code snippet :



We use the term *formal parameters* to refer to the parameters in the definition of the method. In the example, *x* and *y* are the *formal parameters*. You can remember to call them "*formal*" because they are part of the method's definition, and you can think of a definition as being formal.

We use the term *actual parameters* to refer to variables in the method call, in this case *length* and *width* are the *actual parameters*. They are called "*actual*" because they determine the actual values that are sent to the method.

8.4.2 Arguments to Functions

When you pass arguments to functions, you can pass any value of a legal Java datatype. That is, arguments to functions can be :

- (i) of primitive datatypes i.e., char, byte, short, int, long, float, double, boolean.
- (ii) of reference datatypes i.e., objects or arrays.

A function is invoked in two manners : Call by Value and Call by Reference. Basically these two ways of invoking functions are also referred to as Pass by Value and Pass by Reference as these depict the ways arguments are passed to functions. In the following two sections we are going to explore these two ways of passing arguments.

Note

To run and test a function, you may create a class within which write the function to be tested. Also write `main()` method in the same class and invoke the function to be tested inside `main()`. But remember one thing, `main()` is created as static function, hence it can directly invoke only those functions which are static. Other functions are to be invoked through objects i.e., as `<objectname> <functionname>` parameters. If you do not want to create a separate object, you can do so by creating a temporary object and calling the desired function as `new<classname>().<methodname>(parameters)`.

Carefully go through the example programs given in this chapter. Solved problem 11 of Section B uses this very concept.

8.5 Pass By Value (Call By Value)

The call by value method copies the values of *actual parameters*¹ into the *formal parameters*², that is, the function creates its own copy of argument values and then uses them. To understand this concept, let us consider one example.

To test your grammar, your English Teacher purposely writes grammatically incorrect passage on her sheet and gives it to you for corrections. So you copy down the given passage on your own sheet and make corrections there.

This is an example of **call by value** method. Here, your teacher passes the sheet having grammatically incorrect passage (which is the actual parameter here) to you, you copy the passage on your sheet (this is the formal parameter). Whatever changes take place, are not reflected back to the original as the value of original is copied onto another which is the work copy.

Thus, *in call by value method, the changes are not reflected back to the original values*. Following Fig. 8.3 illustrates it.

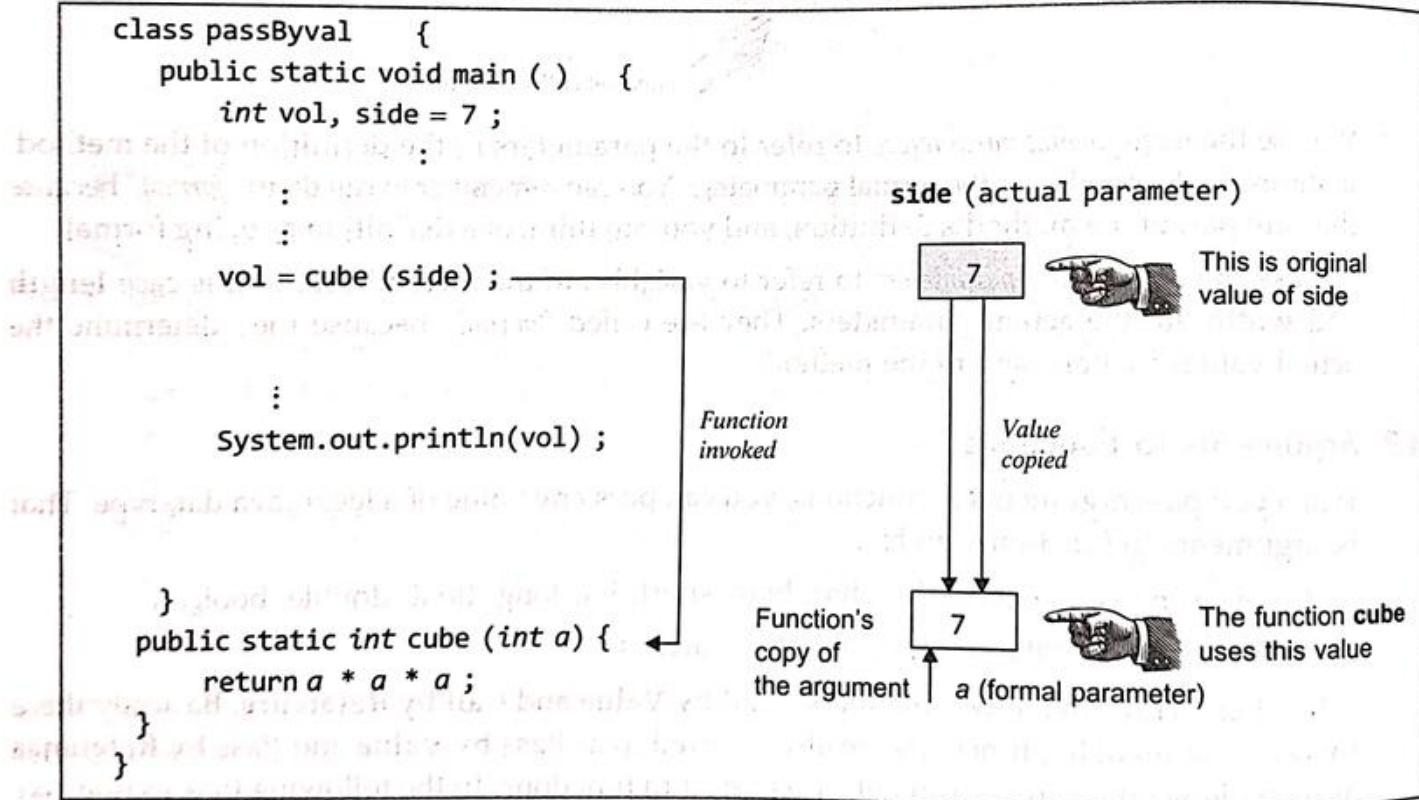


Figure 8.3 Call by Value Method.

The main benefit of call by value method is that you cannot alter the variables that are used to call the function because any change that occurs inside function is on the function's copy of the argument value. The original copy of the argument value remains intact. Following program (8.2) illustrates the call by value method.

1. The parameters that appear in a function call statement are **actual parameters**.
2. The parameters that appear in function definition are **formal parameters**.

Program 8.2

Program to illustrate the call by value method of function invoking.

```
public class prg13_2 {
    public static void main(String args[ ]) {
        int orig = 10 ;
        System.out.println("The original value is:" + orig) ;
        System.out.println("The value after function change( ) is over is:" + orig) ;
    }

    public static int change(int a) {
        a = 20 ; //the value of a is changed inside function changed
        System.out.println ("Inside method change( ), value is now changed to" + a) ;
        return a ;
    }
}
```

The original value is : 10

Inside method change(), value is now changed to 20

The value after function change() is over is : 10

In this example program, the value of the argument to change(), *orig i.e., 10* is copied onto parameter *a* so that *a* gets value 10. When the statement *a = 20* takes place, the value of *a* is changed but not the value of *orig*. The *orig* has still the value 10. Hence the output is as shown above.

Remember that it is copy of the value of the argument that is passed to into the function. What occurs inside the function has no effect on the variable used in the function call.

8.6 Call By Reference

In call by value method, the called function creates a new set of variables and copies the values of arguments into them. The function does not have access to the original variables (actual parameters) and can only work on the copies of values it created. Passing arguments by value is useful when the original values are not to be modified. In fact, call by value method offers insurance that the function cannot harm the original value.

The call by reference method uses a different mechanism. In place of passing a value to the function being called, a *reference* to the original variable is passed. Remember that a *reference* stores a memory location of a variable.

Provision of the reference variables in Java permits us to pass parameters to the function by reference. When a function is called by reference, then, the *formal parameters* become *references* to the *actual parameters* in the calling function³. This means that in the call by reference method, the called function does not create its own copy of original values, rather, it refers to the original values only by different names *i.e.,* the references. Thus the called function

Note During call by value, any change in the formal parameter is not reflected back to actual parameter.

3. A *calling function* is the function that calls another function and the function being called is, the *called function*.

works with the original data and any change in the values gets reflected to the data. To understand this concept, let us revise our same old example of grammatically incorrect passage.

If your English teacher gives you the original sheet having the grammatically incorrect passage and allows you to work upon the same sheet, then whatever corrections you make, will be there on the original. In other words, I can say the changes are reflected back to the original as the value of the original is not copied anywhere rather original itself has been made the work copy.

Thus, in call by reference method, the changes are reflected back to the original values.

The call by reference method is useful in situations where the values of the original variables are to be changed using a function. Say, for instance, a function is to be invoked that swaps the values of two int variables.

If we try doing this through call by value mechanism by passing two int variables then values will be swapped in the work copy of the called function, but not in the original variable. But with call by reference, this can be achieved by passing an object (the reference type) containing two integers whose values are to be swapped. Program 8.3 illustrates these two ways of passing parameters.

Note In Java, all primitive types are passed by value and all reference types (objects, arrays) are passed by reference.

Program 8.3

Swap values of two integers through both types of function calling mechanism.

```
public class prg13_3 {
    public static int x = 10 ;
    public static int y = 20 ;
    public static void main(String args[ ]) {
        System.out.println("Values initially. x =" + x + ", y =" + y) ;
        swapCallByValue(x, y) ;

        System.out.println("Values after swapCallByValue function. x =" + x + ", y =" + y) ;
        prg13_3 object1 = new prg13_3( ) ;           //an object created
        swapCallByRef(object1) ;                     //object passed
        System.out.println("Values after swapCallByRef function. x =" + x + ", y =" + y) ;
    }
    //in call by value, Java primitive types are passed
    public static void swapCallByValue(int a, int b) {
        int tmp ;
        tmp = a ;
        a = b ;
        b = tmp ;
        System.out.println("Values inside swapCallByValue function. x =" + a + ", y =" + b) ;
    }
}
```

```
//in call by ref., reference type is passed
public static void swapCallByRef(prg13_3 obj) {
    int tmp ;
    tmp = obj.x ;
    obj.x = obj.y ;
    obj.y = tmp ;
    System.out.println("Values inside swapCallByRef function. x =" + obj.x + ", y =" + obj.y) ;
}
```

The output produced by above program given below :

```
values initially. x =10, y =20
values inside swapCallByValue function. x =20, y =10
values after swapCallByValue function. x =10, y =20
values inside swapCallByRef function. x =20, y =10
values after swapCallByRef function. x =20, y =10
```

Internally, the memory status for the two called functions of program 8.3 would be somewhat as shown in Fig. 8.4.

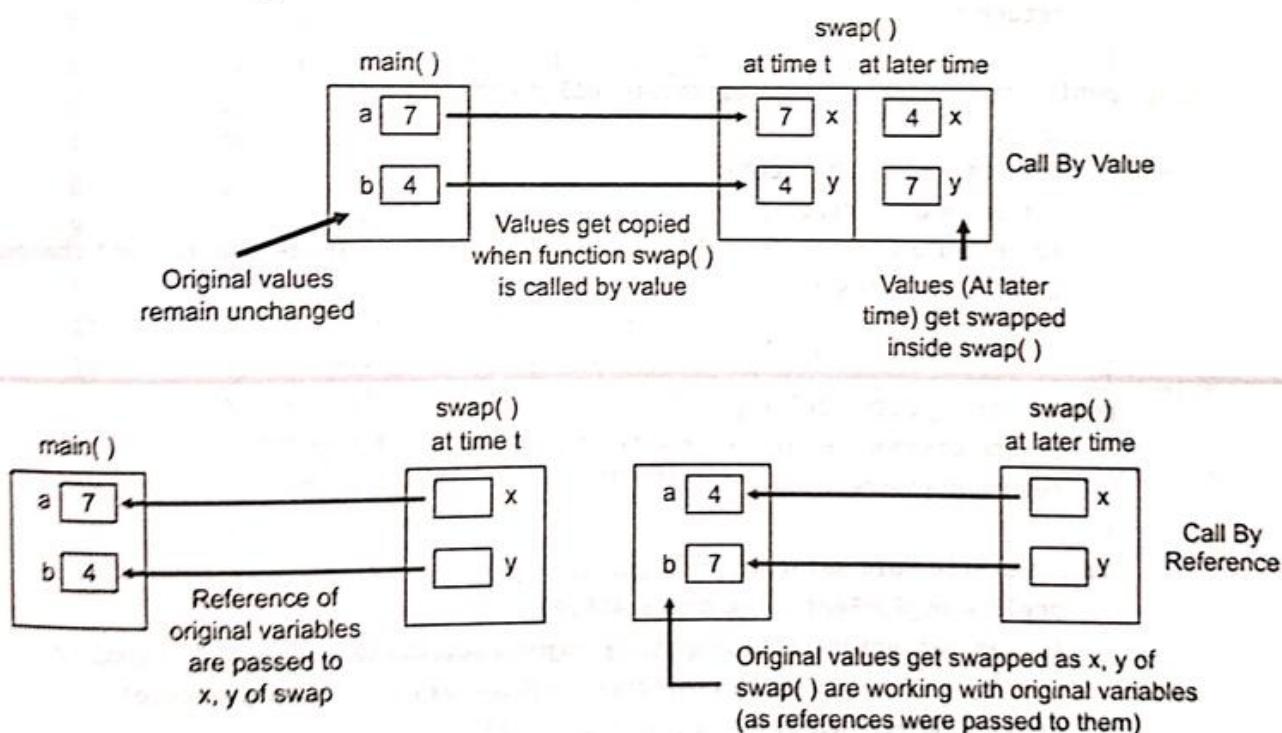


Figure 8.4 Difference between call by value and call by Reference.

After this, let us call by reference methodology once again for further discussion.

Program 8.4

Program to convert distance in feet or inches using a call by reference method.

```
public class prg13_4 {
    private static float feet ;
    private static int inches ;
```

```

static String convertedTo ;
public prg13_4( ) {
    feet = 0 ;
    inches = 0 ;
    convertedTo = "empty" ;
}
public prg13_4(float f, int i) {
    feet = f ;
    inches = i ;
    convertedTo = "not changed yet" ;
}
public static float convertToFeet(prg13_4 obj) {
    int y = obj.inches ;
    float f = y ;
    int fts = y/12 ;
    int ins = (int)(f - fts * 12) ;
    f = fts ;
    f += (float)ins/10 ;
    convertedTo = "feet" ; //Coz of Call by reference this member will get changed
    return f ;
}
public static int convertToInches(prg13_4 obj) {
    int i ;
    int fts = (int)obj.feet ;
    int ins = obj.inches ;
    convertedTo = "inches" ; //Coz of Call by reference this member will get changed
    i = fts * 12 +ins ;
    return i ;
}
public String Distance( ) {
    String distance = (int)feet + "\' " + inches + "\" " ;
    return distance ;
}
public static void main(String args[ ]) {
    prg13_4 objForFeet = new prg13_4(5,2) ;
    System.out.println("Distance " + objForFeet.Distance( ) + " is equal to " +
        convertToInches(objForFeet) + " " + convertedTo) ;
    prg13_4 objForInches = new prg13_4(0, 16) ;
    System.out.println("Distance " + objForInches.Distance( ) + " is equal to " +
        convertToFeet(objForInches) + " " + convertedTo) ;
}

```

Distance 5' 2" is equal to 62 inches
 Distance 0' 16" is equal to 1.4 feet

In both cases a copy of the variable is passed to the method. It is a copy of the *value* for a primitive data type variable ; it is a copy of the *reference* for an object variable. So, a method

receiving an *object variable* as an argument receives a copy of the reference to the original object. Here's the clincher : If the method uses that reference to make changes to the object, then the original object is changed. This is reasonable because both the original reference and the copy of the reference refer to same thing — *the original object*.

There is one exception : *strings*. Since String objects are immutable in Java, a method that is passed a reference to a String object cannot change the original object. This distinction is also brought out in this section.

Let's explore pass by reference and pass by value once again through a demo program (see Program 8.5). As a self-test, try to determine the output of this program on your own before proceeding.

Note In Java, *String* objects are passed by reference but cannot be changed. To make changes in original object, *StringBuffer* type objects should be passed.

Program 8.5

Program to illustrate the differences between two ways of invoking methods and passing parameters.

```

1  public class prg13_5 {
2      public static void main(String[ ] args) {
3          // Part I - primitive datatypes
4          int i = 25 ;
5          System.out.println("int i being passed") ;           //print it (1)
6          System.out.println("Original i :" + i) ;
7          iMethod(i) ;                                     //print it (3)
8          System.out.println("Back in main : " + i) ;
9          System.out.println("-----") ;
10         // Part II - objects and object references
11         StringBuffer sb = new StringBuffer("Hello, world") ;
12         System.out.println("Object being passed") ;
13         System.out.println("Original object :" + sb) ;       //print it (4)
14         sbMethod(sb) ;                                     // print it (6)
15         System.out.println("Back in main( ) :" + sb) ;
16         System.out.println("-----") ;
17         // Part III - strings
18         String s = "Java is fun!" ;
19         System.out.println("String being passed") ;           //print it (7)
20         System.out.println(s) ;
21         sMethod(s) ;                                     //print it (9)
22         System.out.println("Back in main( ) :" + s) ;
23     }
24     public static void iMethod(int iTest) {
25         iTest = 9 ;                                         //change it
26         System.out.println("Value in called method :" + iTest) ; //print it (2)
27     }
28     public static void sbMethod(StringBuffer sbTest) {
29         // function insert inserts Java at 7th position
30         sbTest = sbTest.insert(7, "Java") ;                 //change it

```

```

31     System.out.println("Value in called method :" + sbTest) ; //print it (5)
32   }
33   public static void sMethod(String sTest) {
34     // function substring extracts substring of
35     // 11 characters from 8th character onwards
36     sTest = sTest.substring(8, 11) ; //change it
37     System.out.println("Value in called method :" + sTest) ; //print it (8)
38   }
39 }

```

This program generates the following output :

```

int i being passed
original i : 25
value in called method : 9
Back in main : 25

```

```

Object being passed
original object : Hello, world
value in called method : Hello, Javaworld
Back in main() : Hello, Javaworld

```

```

String being passed
Java is fun!
value in called method : fun
Back in main() : Java is fun!

```

The class *DemoCallingMethods* is organized in three parts. The first deals with primitive datatypes, which are passed by value. The program begins by declaring an int variable named *i* and initializing it with the value 25 (line 4). The memory assignment just after line 4 is illustrated in Fig. 8.5(a). The value of *i* is printed in line 6.

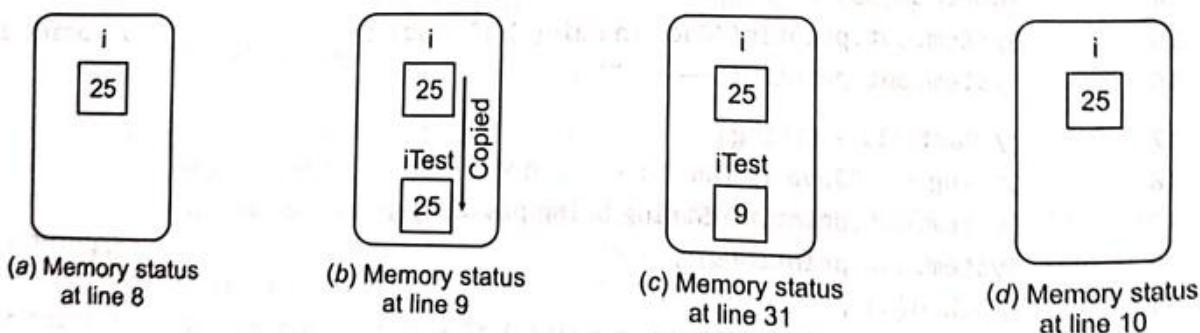


Figure 8.5 Memory assignments for call-by-value example (int variables).

Then, the int variable is passed as an argument to a method called *iMethod()* (line 7). The method is defined in lines 24-27. Within the method, a copy of *i* exists as a local variable *iTest*. The memory assignments just after *iMethod()* begins execution are shown in Fig. 8.5. Note that *i* and *iTest* are distinct : They are two separate variables that happen to hold the same value. Within the method, the value is changed to 9 (line 25) and then printed again.

Back in the `main()` method the original variable is also printed again (line 8). Changing the passed value in `iMethod()` had no effect on the original variable, and, so, the third value printed is the same as the first.

Part II of the program deals with objects and object references. The pass-by-reference concept is illustrated by the object variables `sb` and `sbTest`. In the `main()` method, a `StringBuffer` object is instantiated and initialized with "Hello, world" and a reference to it is assigned to the `StringBuffer` object variable `sb` (line 11). `StringBuffer` is a separate class and its object is not treated as a `String` type object.

The memory assignments for the object and the object variable are illustrated in Fig. 8.6(a). This corresponds to the state of the `sb` just after line 11 in Program 8.5. The object is printed in line 13. In line 14, the `sbMethod()` method is called with `sb` as an argument. The method is defined in lines 28-32. Within the method, a copy of `sb` exists as a local variable named `sbTest`. The memory assignments just after the `sbMethod()` begins execution are shown in Fig. 8.6(b). Note that `sb` and `sbTest` refer to the same object.

In line 30, the object is modified using the `insert()` method of the `StringBuffer` class. The method is called through the instance variable `sbTest`. The memory assignments just after line 30 execute are shown in Fig. 8.6(c). After the `sbMethod()` finishes execution, control passes back to the `main()` method and `sbTest` ceases to exist. The memory assignments just after line 14 in Program 8.5 are shown in Fig. 8.6(d). Note that the original object has changed, and that the original object variable, `sb`, now refers to the updated object.

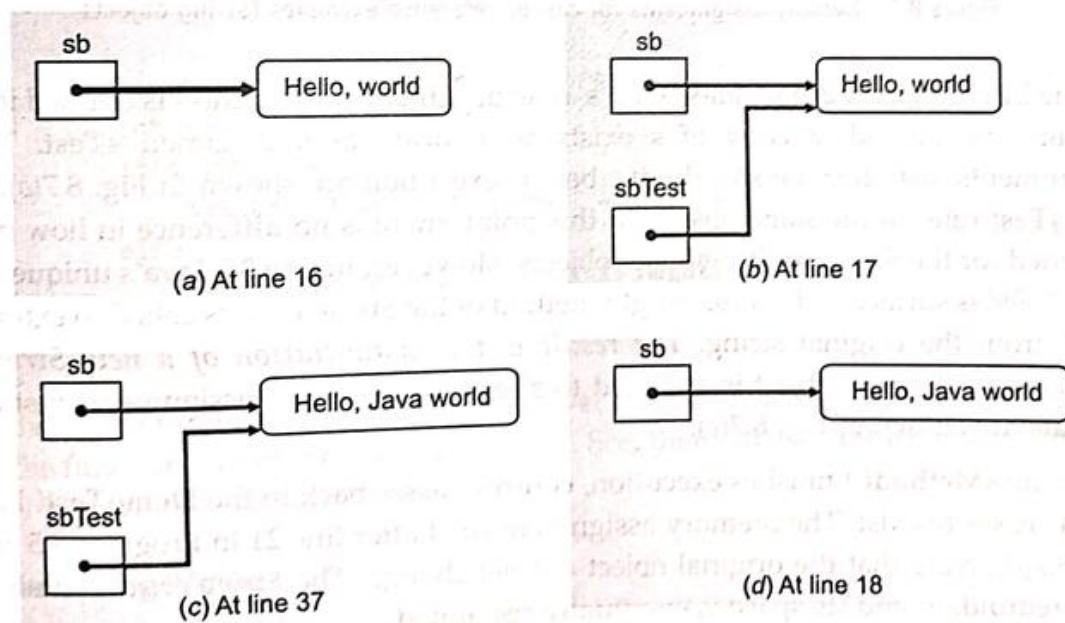
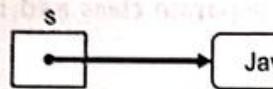


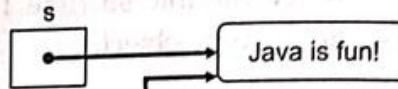
Figure 8.6 Memory assignments for call-by-reference example (StringBuffer Objects).

The scenario played out above, is a typical example of pass by reference. It demonstrates that methods can change the objects instantiated in other methods when they are passed a reference to the object as an argument. A similar scenario could be crafted for objects of any of Java's numerous classes. The `StringBuffer` class is a good choice for the example, because `StringBuffer` objects are tangible, printable entities.

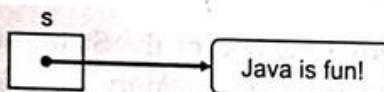
Part III of the program deals with strings. The *String* class is unlike other classes in Java because *String objects*, once instantiated, are immutable i.e., cannot change. For completeness, let's walk through the memory assignments for the *String* objects in the *DemoCallingMethods* program. A *String object* is instantiated in line 18 and a reference to it is assigned to the *String* object variable *s*. The memory assignments at this point are shown in Fig. 8.7(a).



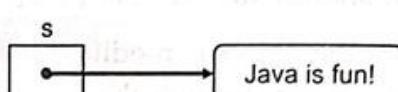
(a) At line 22



(b) At line 25



(c) At line 43



(d) At line 26

Figure 8.7 Memory assignments for call-by-reference examples (String objects).

In line 21 the *sMethod()* is called with *s* as an argument. The method is defined in lines 33-38. Within the method, a copy of *s* exists as a local variable named *sTest*. The memory assignments just after the *sMethod()* begins execution are shown in Fig. 8.7(b). Note that *s* and *sTest* refer to the same object. At this point, there is no difference in how memory was assigned for the *String* or *StringBuffer* objects. However, in line 36, Java's unique treatment of *String objects* surfaces. The *substring()* method of the *String* class is called to extract the string "fun" from the original string. *The result is the instantiation of a new String object.* A reference to the new object is assigned to *sTest*. The memory assignments just after line 36 execute are shown in Fig. 8.7(c).

After the *sMethod()* finishes execution, control passes back to the *DemoTest()* method and *sTest* ceases to exist. The memory assignments just after line 21 in Program 8.5 are shown in Fig. 8.7(d). Note that the original object did *not* change. The *String object* containing "fun" is now redundant and its space is eventually reclaimed.

8.7 Returning from a Function

As invoking a function is important, returning from a function is equally important as it not only terminates the function's execution but also passes the control back to the calling function. A function terminates when either a *return* statement is encountered or the last statement in the function is executed. Generally, a *return* statement is used to terminate a function whether or not it returns a value.

8.7.1 The return statement

The **return** statement is useful in two ways. First, an immediate exit from the function is caused as soon as a **return** statement is encountered and the control passes back to the operating system which is main's caller. Second use of **return** statement is that it is used to return a value to the calling code.

Even though it is not necessary to have a **return** statement in a function, most functions rely on the **return** statement to stop execution either because a value must be returned or to make a function's code simpler and more efficient.

A function may contain several **return** statements. However, only one of them gets executed because the execution of the function terminates as soon as a **return** is encountered. Following program uses multiple **return** statements.

Program 8.6

Program to check whether a given number is positive or not. It passes the number to a function which returns true if it is positive otherwise returns false.

```
public class prg13_6 {
    public static boolean positive(int y) {
        if(y >= 0)
            return true;
        else
            return false;
    }

    public static void main(String[ ] args) {
        int x = 5;
        if(positive(x))
            System.out.println("Number is positive");
        else
            System.out.println("Number is negative");
    }
}
```



Multiple return statements

The output produced is
Number is positive

In the above program, the function **Positive()** checks whether given integer is > 0 or not. If it is, the function *true* otherwise returns *false*. See, there are two **return** statements out of which only one will get executed.

8.7.2 Returning Values

All functions except those of type **void**, return a value. This value is explicitly specified by the **return** statement. If in a function, a **return** statement is missing, then the return value of the function is technically undefined. Only the functions declared as **void** do not return a value and hence cannot be used in expressions. There may be *three* types of functions in Java :

1. **Computational Functions.** The functions that calculate or compute some value and return the computed value. For example, **Math.sqrt()** and **Math.cos()** are computational functions. Computational functions always return a computed result.

Note A function can return only a single value.

2. Manipulative Functions. The functions that manipulate information and return a success or failure code. Generally, if value 0 is returned, it denotes successful operation ; any other number denotes failure.

3. Procedural Functions. The functions that perform an action and have no explicit return value. For instance, `System.out.println()` function is a procedural function.

The functions returning no value should be declared as returning type `void` so that its accidental misuse in expression(s) can be prevented.

A value is returned from a function using `return` statement as it is shown below :

```
return value ;
```

where `value` is a value of the same data type as that of the return data type of the function.

All functions perform one or the other job, however, every function might not return a value. To understand this, let us take one example.

Note For all the functions, their prototypes must be declared. n return only a single value.

You are renovating your house. You employ two people for work – one for stitching curtains for your house and other one for cleaning the house. The first person stitches the curtains and hands over beautifully stitched curtains to you. The second person also performs his job well and cleans the house thoroughly. He, however, has not anything tangible to give you back. Now, in this case, both the workers have performed their duties well. But the first one has returned a value whereas second one has not.

Same way, all functions perform their assigned actions, however, some return values, some don't. For instance, `println()` function performs its job i.e., prints something but it does not have any thing to return. On the other hand, `sqrt()` function returns the square root of the number passed to it. The functions returning values make use of `return` statement to return the value.

8.8 Pure and Impure Functions

The methods that you write or create can either be a pure function or impure function. A *pure function* is the one that takes objects and/or primitives as arguments but does not modify the objects. The return value of a pure function is either a primitive or a new object created inside the method. An *impure function*, on the other hand, changes/modifies the state of received object.

A method is considered a pure function if the result depends only on the arguments, and it has no side effects like modifying an argument or printing something. The only result of invoking a *pure function* is the return value.

One example function is `after`, which compares two `Time` type objects and returns a `boolean` that indicates whether the first operand comes after the second :

```
public static boolean after(Time time1, Time time2) {
    if (time1.hour > time2.hour) return true ;
    if (time1.hour < time2.hour) return false ;
```

```

if (time1.minute > time2.minute) return true ;
if (time1.minute < time2.minute) return false ;
if (time1.second > time2.second) return true ;
return false ;
}

```

Thus, *pure function* is one without any *side-effects*. A side-effect really means that the function keeps some sort of hidden state inside it. A function or procedure in a programming language is said to have side-effects if it makes changes to its environment, functions that have no side-effects i.e., the *pure functions* can be called any number of times and in any order. It doesn't matter how many times you evaluate the area of a triangle, you will always get the same answer ; but if the function to calculate the area has a side-effect such as changing the size of the triangle, or if you don't know whether it has side-effects or not, then it becomes important to call it once only.

Impure functions (also called *modifier functions*), on the other hand, change the state of the argument received. Following method is an example of an impure function. This method – *increment* method adds a given number of seconds to a *Time* object.

```

public static void increment(Time time, double secs) {
    time.second += secs ;
    if(time.second >= 60.0) {
        time.second -= 60.0 ;
        time.minute += 1 ;
    }
    if(time.minute >= 60) {
        time.minute -= 60 ;
        time.hour += 1 ;
    }
}

```

8.9 Function Overloading

When several function declarations are specified for a single function name in the same scope, the (function) name is said to be overloaded. Java allows functions to have the same name if it can distinguish them by their number and type of arguments. For example, following two functions are different for Java.

```

float divide (int a, int b) { ... }
float divide (float x, float y) { ... }

```

That is, **divide()** taking two **int** arguments is different from **divide()** taking two **float** arguments.

This is known as function overloading.

A function name having several definitions in the same scope that are differentiable by the number or types of their arguments, is said to be an overloaded function.

8.9.1 Need For Function Overloading

Objects have characteristics and associated behaviour. A *bird* object (say a parrot) can see which is its behaviour. But its behaviour may differ in different situations. For instance, if the

message, 'See through daylight', is passed to it, the parrot will accomplish this task i.e., the parrot will be able to see through daylight. But if the message, 'See through darkness' is passed to it, the parrot will not be able to see. That is, the same behaviour of same object may differ in different situations. Then, why can't this happen with function(s) of a class? After all, it is a class that implements OOP in practice. Therefore, in order to simulate real-world objects in programming, it is necessary to have function overloading.

You must be thinking now that when we can define different functions for the different situations, then why should we repeat the same function name again and again? Yes, you are right. Alternatively, you always can define different functions for the different situations as it is shown below:

```

    :
    void see_day( ) {                                // Function for seeing through daylight
        System.out.println ("Can See through Daylight \n") ;
    }
    void see_night( ) {                             // Function for seeing through night
        System.out.println ("Cannot see through darkness \n") ;
    }
}

```

But such multiple definitions, you need to yourself decide upon which function should be executed. For example,

```

if (choice == 'D')
    see_day( ) ;
else if (choice == 'N')
    see_night( ) ;
else
    System.out.println ("\n Wrong choice \n") ;
:

```

Would not it be nice if the compiler automatically decides which function should get executed? It will not only reduce the code by reducing the number of if-else but also make the code execute faster as so many comparisons are eliminated. But, you know, the compiler decides about it only when functions are overloaded.

Note Function Overloading not only implements polymorphism but also reduces number of comparisons in a program and thereby makes the program run faster.

Following figure (Fig. 8.8) illustrates function overloading :

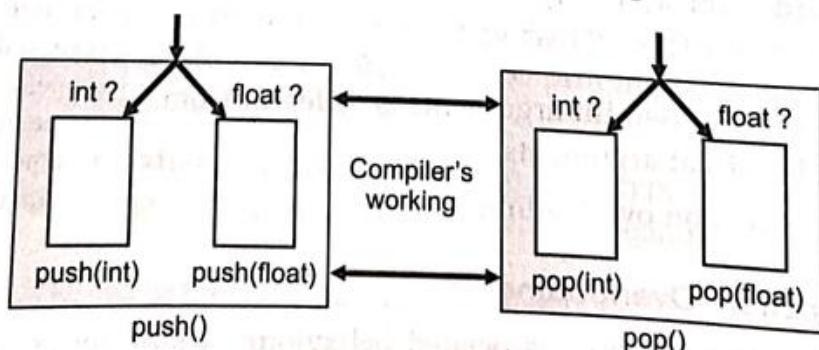


Figure 8.8 Function Overloading.

8.9.2 Declaration and Definition

The key to function overloading is a function's argument list which is also known as the *function signature*. It is the *signature*, not the function type that enables function overloading.

A function's argument list is known as the function's signature.

If two functions are having same number and types of arguments in the same order, they are said to have the same *signature*. Even if they are using distinct variable names, it does not matter. For instance, following two functions have same signature.

```
void squar (int a, float b) ;      // function 1
void squar (int x, float y) ;      // same signature as that of function 1
```

Java lets you overload a function name provided the functions with same name are having different signatures. The signature can differ in the number of arguments or in the type of arguments, or both. To overload a function name, all you need to do is, declare and define all the functions with the same name but different signatures, separately. For instance, following code fragment overloads a function name `prnsqr()`:

```
void prnsqr (int i) {
    System.out.println ("Integer" + i + " 's square is" + i * i) ;
}
void prnsqr (char c) {
    System.out.println (c + " is a character" + "Thus No Square for it") ;
}
void prnsqr (float f) {
    System.out.println ("Float" + f + " 's square is" + f * f) ;
}
void prnsqr (double d) {
    System.out.println ("Double float" + d + " 's square is" + d * d) ;
}
```

Thus, we see that there is not too much difficulty in declaring overloaded functions ; they are declared as other functions are. Just one thing is to be kept in mind that the arguments are sufficiently different to allow the functions to be differentiated in use.

The argument types are said to be part of the function's extended name. For instance, the name of above specified functions might be `prnsqr()` but their extended names are different. That is they have `prnsqr(int)`, `prnsqr(char)`, `prnsqr(float)`, and `prnsqr(double)` extended names respectively.

When a function name is declared more than once in a program, the compiler will interpret the second (and subsequent) declaration(s) as follows :

- (i) if the signatures of subsequent functions match the previous function's, then the second is treated as a re-declaration of the first.
- (ii) if the signatures of the two functions match exactly but the return types differ, the second declaration is treated as an erroneous re-declaration of the first and is flagged at compile time as an error. For example,

```
float square (float f) { ... }
double square (float x) { ... } // error
```

Functions with the same signature and same name but different return types are not allowed in Java. You can have different return types, but only if the signatures are also different :

```
float square (float f) { ... }           // different signatures, hence
double square (double d) { ... }         // allowed
```

- (iii) if the signatures of the two functions differ in either the number or type of their arguments, the two functions are considered to be overloaded.

TIP

Use function overloading only when a function is required to work for alternative argument types and there is a definite way of optimizing the function for the argument type.

Use function overloading only when a function is required to work for alternative argument types and there is a definite way of optimizing the function for the argument type.

8.10 Calling Overloaded Functions

Overloaded functions are called just like other functions. The number and type of arguments determine which function should be invoked. For instance, consider the following code fragment :

```
prnsqr ('z') ;                      // calls # 2
                                         [Refer to section 8.9.2 for definitions]
prnsqr (13) ;                       // calls # 1
prnsqr (134.520000012) ;            // calls # 4
prnsqr (12.5F) ;                   // calls # 3
```

A function call first matches the prototypes available with the number and type of arguments provided with the function call and then calls the appropriate function for execution. But sometimes there might be ambiguity between float and double values or say int or long values. For instance, if you want to invoke the function with following declaration

```
void prnsqr (double d) { ... }
```

with the value 1.24. This value may also be assumed to be float as well as double. Now to avoid such ambiguity, we can use constant suffixes (F, L, D, etc.) to distinguish between such values as these greatly help in indicating which overloaded function should be called.

TIP

When calling functions, use constant suffixes to avoid ambiguity.

An ordinary floating constant (312.32) has the double type, while adding the F suffix (312.32F) makes it a float. The suffix L (312.32L) makes it a long double. Similarly, suffix D or

Let Us Revise

- ❖ A method or a function is a sequence of statements that carry out specific task(s).
- ❖ Functions are helpful in coping with complexity, hiding low-level details and providing reusability.

- ❖ First line of the function definition is function prototype that tells the program about the type of value returned by the function and the number and type of arguments.
- ❖ A function signature refers to the number and types of arguments.
- ❖ Functions not returning any value are declared void functions i.e., having void as their return type.
- ❖ Actual parameters are the parameters appearing in function call statement.
- ❖ Formal parameters are the ones that appear in function definition.
- ❖ Arguments are passed either by value or by reference.
- ❖ In Java, all primitive types are passed by value and all reference types are passed by reference.
- ❖ In pass by value, the called method copies the actual value of the passed variable in its own work copy, which is different from original variable.
- ❖ In pass by reference, the called method copies the reference of passed variable (object or array) and works with original copy.
- ❖ A function returns value through return statement.
- ❖ Pure functions are those that do not change the state of their parameters.
- ❖ Impure functions are those that change/modify the state of their parameters. Impure functions are also called modifier functions.
- ❖ A function having multiple definitions (in the same scope) that are differentiable by their signatures, is called an overloaded function. This process is called function overloading.

8.11 Constructors

A constructor is a member function of a class that is called for initializing when an object is created of that class. It (the constructor) has the same name as that of the class's name and its primary job is to initialize the object to a legal initial value for the class.

If a class has a constructor, each object of that class will be initialized before any use is made of the object. Consider the following class having a constructor :

```
class Student {
    private int rollno ;
    private float marks ;
    public Student( ) { // constructor
        rollno = 0 ; 
        marks = 0.0 ;
    } // other public members
};
```

*The constructor see has same name as that of its class ;
and also, it has no return type, not even void.*

See the above class has a member function with the same name as its class has and which provides initial values to its data members. Nowforth, whenever an object of the **Student** type will be created, the compiler will call the constructor **Student()** for the newly created object.

A member function with the same name as its class is called **Constructor** and it is used to initialize the objects of that class type with a legal initial value.

Note Constructors have no return type, not even void.

8.11.1 Need for Constructors

Constructors have one purpose in life: to create an instance of a class. This can also be called creating an object, as in :

```
Student s1 = new Student();
```

The purpose of methods, by contrast, is much more general. A method's basic role is to execute Java code.

8.11.2 Declaration and Definition

A constructor is a member function of a class with the same name as that of its class name. A constructor is defined like other member functions of a class. Following code fragment defines a constructor :

```
class X {    int i ;
            public int j, k ;
            public X () {
                i = 0 ;
                j = 0 ;
                k = 0 ;      // constructor
            }
            :
            // other members
        } ;
```

*See, this method has
same name as that of
its class*



In the above given class definition, the constructor has been defined as a *public* member function. You even can define a constructor under *private* sections. A constructor also, obeys the usual access rules of a class. That means, a *private* constructor is not available to the non-member functions. In other words, with a *private* constructor, you cannot create an object of the same class in a non-member function, however, this is allowed in the member functions. Following code fragment explains it.

```
class X{
    int i ;
    private X() { i = 10 ; j = 10 ; k = 10 ; }      // constructor defined as private
    public int j, k ;
    public void getval( ) {                            // member function 1
        :
    }
    void check( ) {                                // valid here. The constructor of X can be
        X o1= new X( ) ;                            // accessed by check( ) as it is a member
        :
        // function of class X
    }
}
class Y{
    public static void main(String args[ ]) {
        X o1 = new X( ) ;      // invalid here. The constructor of X cannot be accessed
        :
        // by main( ) as it is a non-member function of class X
    }
}
```

In the above example, since constructor of X is *private*, objects of X can be created only inside member functions, but not inside non-member functions. Reason being that everytime an object is created, the constructor is automatically invoked; but if the function, declaring the object, does not have access privilege for the constructor, it (the constructor) cannot be invoked for the object, thus, the object can not be created under such function. A class having no public constructors is a *private class*. Only the member functions may declare objects of the class.

Note Generally, a constructor should be defined under the public section of a class, so that its objects can be created in any function.

Differences between Constructors and Methods

Though constructors are members of a class just like methods, yet they are different from one another. Following table summarizes the differences between the two :

Parameter	Constructors	Methods
Purpose	Creates an instance of a class	Groups Java statements
Return type	Has no return type , not even void	void or a valid return type
Name	Same name as the class()	Any name except the class. ()
Execution	Called at the time of object creation.	Called when a function call for the specific method is encountered. Method-calls are to be specified by the programmer.

8.12 Types of Constructors

The constructor functions in Java can be of two types. One, those which can receive parameters (**parameterized**) and second, those which can not receive any parameters (**non-parameterized**).

The above mentioned both the constructors do not take any argument. A constructor can also have arguments. For example,

```
class XYZ {
    private int i ;
    public float j ;
    XYZ (int a, float b) {           //constructor taking arguments
        i = a ;
        j = b ;
    }
    ...
}
```

The above defined constructor takes two values : one int and one float to construct values for the newly created object. With such a constructor, the objects will be created as follows :

```
XYZ O1 = new XYZ (2, 13.5F);      // object O1 created
XYZ O2 = new XYZ (4, 19.22F);     // object O2 created
```

with a constructor requiring arguments, objects can be created in a manners as shown above.

8.12.1 Non-Parameterized Constructors

A constructor that accepts no parameter is called the *non-parameterized constructor*. Non-parameterized constructors are considered **default constructors**. In the previous section, **X()** is the default constructor for class **X** since it takes no parameter. However, the constructor **XYZ()** is not a default constructor. With a default constructor, objects are created just the same way as variables of other data types are created. For instance,

```
X O1 = new X();
```

will create the object **O1** of type **X** by invoking the default constructor. But when the constructor accepts some parameters, the initial values must be passed at the time of object creation. Now, here is a related tip for you.



If a class has no explicit constructor defined, the compiler will supply a default constructor.

When a user-defined class does not contain an explicit constructor, the compiler automatically supplies a default constructor, having no arguments. For instance, consider the following code snippet :

```
class A {
    int i;
    public void getval() { ... }
    public void prnval() { ... }
    ...
}
class B {
    public static void main(String args[ ]) {
        A O1= new A();
        O1.getval();
        O1.prnval();
    }
}
```

 Compiler automatically provided
constructor for class A i.e., A()

Having a default constructor simply means that an application can declare instances of the class, since Java requires that whenever an instance of a class is created, its constructor is called.

There can be a default constructor as well as another constructor with arguments for a class, but that's a case of multiple constructors which we are going to cover a little later in the chapter

8.12.2 Parameterized Constructors

As we have already read, a constructor may also take arguments. Such constructors are called *parameterized constructors*. The parameterized constructors allow us to initialize the various data elements of different objects with different values when they are created. This is achieved by passing different values as arguments to the constructor function when the objects are created.



Note The default constructor provided by the compiler does not do anything specific. It initializes the data members by any dummy value.

Following definition shows a parameterized constructor :

```
class ABC {
    int i ;
    float j ;
    char k ;
    public ABC (int a, float b, char c) { //parameterized constructor
        i = a ;
        j = b ;
        k = c ;
    }
}
```

//other members

With a parameterized constructor for a class, one must provide initial values as arguments, otherwise, the compiler will report an error. For example,

```
ABC Obj1 = new ABC( ) ; //Invalid.
```

will cause an error as the constructor for ABC expects three arguments and no argument value has been supplied at the time of object declaration. Therefore, with a parameterized constructor, the initial values must be passed at the time of object creation. This can be done as follows :

```
ABC obj1 = new ABC (13, 11.4F, 'p') ;
```

This statement will create an object **obj1** of type ABC and invoke the constructor of ABC to initialize **obj1** with values 13, 11.4, and 'p'.

Once you declare a constructor with arguments, the default constructor becomes hidden. After this you cannot invoke the default constructor.

So, the another related tip is :

TIP

Declaring a constructor with arguments hides the default constructor.

This means that you must always specify the arguments whenever you declare an instance (object) of the class. Consider the following code snippet.

```
public class Test {
    int ant ;
    public Test (int i) { // constructor with arguments
        ant = i ;
    }
}
public static void main( String args[ ] ) {
    Test object1 = new Test(45) ; //ok, Argument value provided
    Test object2 = new Test( ) ; //Error !! No default constructor available
}
```

8.13 The this Keyword

As soon as you define a class, the member functions are created and placed in the memory space only once. That is, only one copy of member functions is maintained that is shared by all the objects of the class. Only space for data members is allocated separately for each object (see Fig. 8.9).

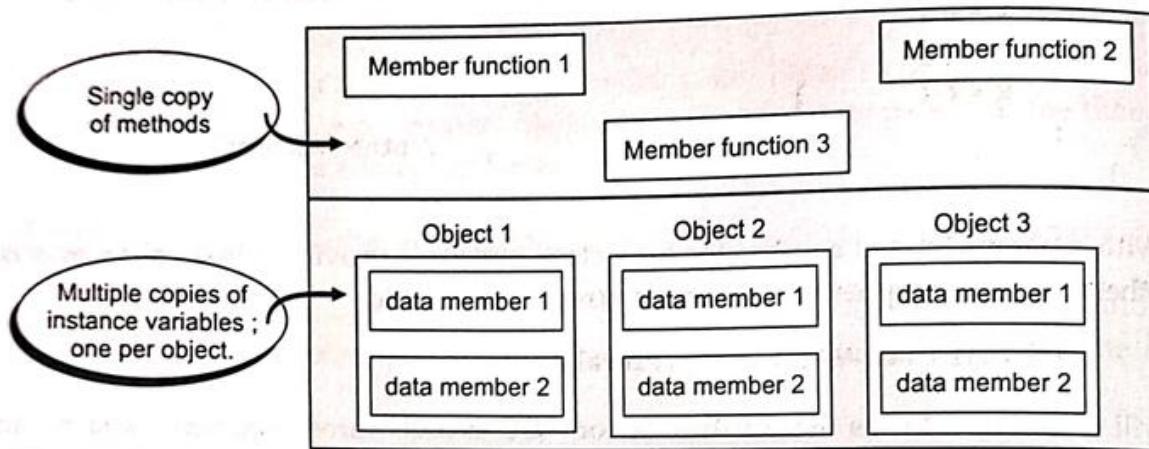


Figure 8.9 Memory allocation for class objects and functions.

This has an associated problem. If only one instance of a member function exists, how does it come to know which object's data member is to be manipulated ? For example, if *member-function3* is capable of changing the value of *datamember2* and we want to change the value of *data-member2* of *object1* how would the *member-function3* come to know which object's *datamember2* is to be changed ?

The answer to this problem is **this** keyword. When a member function is called, it is automatically passed an implicit (in built) argument that is a reference to the object that invoked the function. This reference is called **this**. That is, if *object1* is invoking *member-function3*, then an implicit argument is passed to *member-function3* that points to *object1* i.e., **this** pointer now points to *object1*. The **this** reference can be thought of analogous to the ATM card. For instance, in a bank there are many accounts. The account holders can withdraw amount or view their bank-statements through Automatic-Teller-Machines. Now, these ATMs can withdraw from any account in the bank, but which account are they supposed to work upon ? This is resolved by the ATM card, which gives the identification of user and his account, from where the amount is withdrawn.

Similarly, the **this** reference is the ATM card for objects, which identifies the currently- calling object. The **this pointer** stores the address of currently-calling object. To understand this, consider the following code snippet.

```
// A redundant use of this.
Box(double w, double h, double d) {
    this.width = w ;
    this.height = h ;
    this.depth = d ;
}
```

 *explicit use of this*

The `this` can be implicit or explicit. In implicit `this` Java itself uses keyword `this` to refer to a data member of object e.g., consider following code :

```
Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d; } 
implicit use of this : here width is implicitly treated as  
this.width height : as this.height ; and depth as this.depth
```

The above code will be implicitly converted (using `this` keyword) as :

```
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d; }
```

Or you as programmer can explicitly write above code i.e., use `this` keyword explicitly as shown in above code fragment.

In either case, the `width` property belongs to this `Box` object that assigned `width` to have the value of `w`.

We call the `w`, `h`, and `d`, the formal parameters of the `Box` method, whereas `width`, `height`, and `depth` are the variables of the instance i.e., the object.

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth; } 
see here this is effectively resolving the two widths, two heights and two depths
```

Here, the `this` keyword solves the namespace collision caused by local variable hiding.

What that means is that the formal parameters of your method definition can have the same names as the instance variables of actual object, and yet we can keep the two straight. Consider following code fragment :

```
class Demo {
    int a; // instance variable a declared
    void funcTest() {
        int a; // local variable a declared. Here it hides the instance variable a.
        a = 10; // value 10 assigned to local a.
        this.a = 20; // value 10 assigned to instance variable a.
        // see above statement resolved the name space collision by using keyword this
    }
}
```

The keyword `this`

- ◆ only pertains to an instance method, not to a class method
- ◆ is an implicit argument to this instance of the class
- ◆ is accessible inside of any instance method

Consider the following class that illustrates the usage of **this** keyword.

```
public class Car
{
    private String make = "Ferrari";
    private String model = "F-50";
    private String year = "2007";

    public void copy(Car carToCopy)
    {
        // here we use the this keyword to access the instance variables
        this.make = carToCopy.make;
        this.model = carToCopy.model;
        year = carToCopy.year;           // year is implicitly this.year
    }
}
```

Passed object's variables copied to current object's variables




You can also use **this** to make a method call

(`'this.myMethod()'` is equivalent to `'myMethod()'` in the context of the current object).

The **this** keyword is rarely mandatory, and is for human readability, so that your reader (or yourself) knows right away on which instance the call is made.

8.14 Temporary Instances

The explicit call to a constructor also allows you to create a *temporary instance* or *temporary object*. A *temporary instance* is the one that *lives* in the memory as long as it is being used or referenced in an expression and after this it dies. The temporary instances are anonymous i.e. they do not bear a name.

TIP

An explicit call to the constructor lets you create a temporary instance.

Following example code snippet illustrates it.

```
class Sample {
    int i, j;
    Sample (int a, int b)
    { i = a; j = b; }
    public static void print( )
    {
        System.out.println ( i + " , " + j );
    }
}
public static void main( String[ ] args) {
    Sample S1 = new Sample (2, 5);           //An object S1 created
    S1.print();                            //Data values of S1 printed
    new Sample(4, 9).print();              //Data values of a temporary
                                         // sample instance printed
}

```

Object S1 invokes the print() method



In this example, `sample(4, 9)` is an anonymous temporary instance and lives in the memory as long the statement `sample(4, 9).print();` is executing. The compiler deletes it after use. So here is a tip for you now.

TIP

The temporary instances are deleted when they are no longer referenced.

8.15 Constructor Overloading

Just like any other function, the constructor of a class may also be overloaded so that even with different number and types of initial values, an object may still be initialized. For example, consider the following code fragment that shows a legal class definition, which has overloaded constructors.

```
class ConsOverLoad {
    int a, b ;
    float c ;

    public ConsOverLoad() <----- // #1 non-parameterized constructor
    {
        a = 0 ;
        b = 0 ;
        c = 0 ;
    }

    public ConsOverLoad(int x) <----- // #2 a parameterized constructor
    {
        a = x ;
        b = x ;
        c = 0 ;
    }

    public ConsOverLoad(int x, int y, float z) // #3 another parameterized constructor
    {
        a = x ;
        b = y ;
        c = z ;
    }

    public static void main(String[ ] args)
    {
        ConsOverLoad 01 = new ConsOverLoad( ) ; // Constructor #1 used
        ConsOverLoad 02 = new ConsOverLoad(3) ; // Constructor #2 used
        ConsOverLoad 03 = new ConsOverLoad(3, 5, 7.5F) ; // Constructor #3 used
    }
}
```

Three constructors (overloaded)

Matches with a constructor (#1) that takes no parameter

Matches with a constructor (#2) taking one parameter

Matches with a constructor (#3) that takes three parameters

L et Us Revise

- ❖ A constructor is a member function with the same name as that of its class and it is used to initialize the objects of that class type with a legal initial value.
- ❖ Constructors are invoked by the compiler through new constructor-name().
- ❖ If a class has no constructor defined, the compiler automatically generates one.
- ❖ Constructors can either be non-parameterized or parameterized.
- ❖ A constructor taking no arguments is a non-parameterized default constructor.
- ❖ A constructor receiving arguments is known as parameterized constructor.
- ❖ The this keyword is used to refer to currently calling object.

8.16 Recursive Functions

In many of your programs, you must have written some functions that **call** or **invoke** other functions, e.g.,

```
void A( ) {  
    :  
    B( );           // calling another function  
}
```

Now, if you write a function definition that **calls itself**, then this function would be known as **recursive function**. That is, a function is said to be recursive if the function definition includes a call to itself e.g.,

```
void recur( ) {  
    :  
    recur( );      // calling itself  
}
```

A function is said to be **Recursive Function** if it calls itself.

Recursion can be indirect also. The above example, where a function calls itself directly from within its body is an example of *direct recursion*. However, if a function calls another function which calls its caller function from within its body, it is known as *indirect recursion* e.g.,

```
A( ) {  
    B( );  
}  
B( ) {  
    A( );  
}
```

The above shown sample code is an example of *indirect recursion*. In mathematics also, you have seen recursive problems e.g., sum of n natural numbers can be written as

$$\text{sum of } n \text{ natural numbers} = n + \text{sum of } n-1 \text{ natural numbers}$$

Similarly, $\text{sum of } n-1 \text{ natural numbers} = n-1 + \text{sum of } n-2 \text{ natural numbers}$

Similarly, $\text{sum of } n-2 \text{ natural numbers} = n-2 + \text{sum of } n-3 \text{ natural numbers}$,

and so on.

This section is going to discuss recursion and recursive functions. We shall be learning how to write recursive functions and implement recursion.

8.17 Recursion in Java

As it is mentioned, recursion occurs when a function calls itself. In Java, as in other programming languages that support it, recursion is used as a *form of repetition that does not involve iteration*.

A **Recursive Definition** is a definition that is made in terms of a smaller version of itself. Have a look at following definition of x^n , which is non-recursive :

$$x^n = x * x * \dots * x \quad (\text{Iterative definition - nonrecursive})$$

Now, it can be represented in terms of recursive definition as follows :

$$\begin{aligned} x^n &= x * (x^{n-1}) && \text{for } n > 1 \\ &= x && \text{for } n = 1 \\ &= 1 && \text{for } n = 0 \end{aligned} \quad (\text{Recursive definition})$$

Writing a Recursive Function. Before you start working recursive functions, you must know that every recursive function must have at least *two cases* :

- (i) the **Recursive Case** (or the inductive case)
- (ii) the **Base Case** (or the stopping case)

The **Base Case** is a small problem that we know how to solve and is the case that causes the recursion to end. In other words, it is the case whose solution is pre-known and used without computation.

The **Recursive Case** is the more general case of the problem we're trying to solve using recursive function.

As an example, with the power function x^n , the **recursive case** would be :

$$\text{Power}(x, n) = x * \text{Power}(x, n - 1)$$

and the **base cases** would be

$$\text{Power}(x, n) = x \text{ when } n = 1$$

and

$$\text{Power}(x, n) = 1 \text{ when } n = 0$$

Note Every recursive function consists of one or more base cases and a general, recursive case.

Other cases (when $n < 0$) we are ignoring for now for simplicity sake.

Consider the following example (program 8.7) of Power Function :

Program 8.7

Program to show the use of recursion in calculation of power e.g., a^b

```
import java.io.*;
class PowerRec {
```

```

public static void main(String args[ ] ) {
    PowerRec ob = new PowerRec( );
    long a , b , res ;
    try {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter the value of a : " );
        a = Long.parseLong( br.readLine ( ) );
        System.out.print("Enter the value of b : " );
        b = Long.parseLong( br.readLine ( ) );
        res = ob.pow( a , b );
        System.out.println(a + " ^ " + b + " : " + res );
    }
    catch(Exception e) {
        System.out.println(e);
    }
}

private long pow( long m , long n ) {
    if(n==1) {
        return m;
    }
    else {
        return ( m * pow ( m , n - 1 ) );
    }
}

```

The output produced is

Enter the value of a : 4
 Enter the value of b : 3
 $4 \wedge 3 : 64$

If there is no base case, or if the base case is never executed, an infinite recursion occurs. An Infinite Recursion is when a recursive function calls itself endlessly.

The Power function can be coded as a non-recursive (iterative) function as follows :

```

//Power iterative function

int Power (int x, int n) {
    int pow = 1 ;
    for ( ; n > 0 ; n--)
        pow *= x ;
    return pow ;
}

```

An Infinite Recursion is
 when a recursive function
 calls itself endlessly.

Now, consider another example. The factorial of a non-negative number is defined as the product of all the values from 1 to the number :

$$n! = 1 * 2 * \dots * n$$

It can also be defined recursively as

$$\begin{aligned}
 n! &= 1 && \text{if } n < 2 \\
 &= n * (n-1)! && \text{if } n \geq 2
 \end{aligned}$$

Recursively factorial function would be written as follows :

```
// Factorial Function
int Factorial ( int n ) {
    if ( n < 2 )
        return 1 ;
    return n * Factorial(n - 1) ;
}
```

The factorial function can also be defined iteratively by a simple function as follows :

```
int Factorial ( int n ) {
    int fact = 1 ;
    for( int i = n ; i >= 0 ; i-- )
        fact *= i ;
    return fact ;
}
```

Many functions are much easier to define recursively than they are iteratively.

Note If there is no base case, or if the base case is never executed, an infinite recursion occurs. An Infinite Recursion is when a recursive function calls itself endlessly.

Computing GCD recursively

We can efficiently compute the gcd using the following property, which holds for positive integers p and q :

If $p > q$,

the gcd of p and q is the same as the gcd of p and $p \% q$.

That is, our Java code to compute GCD recursively would be :

```
public static int gcd(int p, int q) {
    if (q == 0) return p;
    return gcd(q, p % q);
}
```

The **base case** is when q is 0, with $\text{gcd}(p, 0) = p$. To see that the reduction step converges to the base case, observe that the second input strictly decreases in each recursive call since $p \% q < q$. If $p < q$ the first recursive call switches the arguments.

```
gcd(1440, 408)
gcd(408, 216)
gcd(216, 24)
gcd(192, 24)
gcd(192, 24)
gcd(24, 0)
return 24
return 24
return 24
return 24
return 24
```

This recursive solution to the problem of computing the greatest common divisor is known as *Euclid's algorithm* and is one of the oldest known algorithms – it is over 2000 years old.

Recursion vs. Iteration

When a loop repeats, it uses same memory locations for variables and repeats the same unit of code. Whereas in recursion, instead of repeating the same unit of code and using the same memory locations for variables, fresh memory space is allocated for each recursive call.

As it happens, any problem that can be solved via iteration can be solved using recursion and any problem that can be solved via recursion can be solved using iteration. Iteration is preferred by programmers for most recurring events, reserving recursion for instances where the programming solution would be greatly simplified. In a programming language, recursion involves an additional cost in terms of the space used in RAM by each recursive call to a function and in time used by the function call.

Because of extra memory stack manipulation, recursive versions of functions often run slower and use more memory than their iterative counterparts. But this is not always the case, and recursion can sometimes make code easier to understand.



Let Us Revise

- ❖ A function is said to be recursive if it calls itself.
- ❖ There are two cases in each recursive function the recursive case and the base case.
- ❖ The base case is the case whose solution is pre-known and is used without computation.
- ❖ The recursive case is more general case of problem, which is being solved.
- ❖ An infinite recursion is when a recursive function calls itself endlessly.
- ❖ If there is no base case, or if the base case is never executed, infinite recursion occurs.
- ❖ Iteration uses same memory space for each pass contrary to recursion where fresh memory is allocated for each successive call.
- ❖ Recursive functions are relatively slower than their iterative counterparts.



Solved Problems

1. Write advantages of using functions in programs.

Solution. Major advantages of using functions are :

- (i) functions lessen the complexity of programs
- (ii) functions hide the implementation details
- (iii) functions enhance reusability of code.

2. Differentiate between Call by Value and Call by Reference.

Solution. In Call by Value, the called function creates its own workcopy for the passed parameters and copies the passed values in it. Any changes that take place remain in the work copy and the original data remains intact.

In Call by Reference, the called function receives the reference to the passed parameters and through this reference, it accesses the original data. Any changes that take place are reflected in the original data.

3. How are following passed in Java ?

- (i) primitive types (ii) reference types

Solution. (i) By value, (ii) By reference.

4. The String objects being reference types are passed by reference but changes, if any, are not reflected back to them. Why ?

Solution. The String objects are immutable in Java, which means once they are created, they cannot be changed. That is why, even though Strings are passed by reference, they cannot be changed.

5. Consider the following class :

```
public class IdentifyMyParts {
    public static int x = 7 ;
    public int y = 3 ;
}
```

(a) How many class variables does the IdentifyMyParts class contain ? What are their names ?

(b) How many instance variables does the IdentifyMyParts class contain ? What are their names ?

Solution. (a) Only 1, namely x (b) Only 1, namely y.

6. Define a method called sayHello in class Methodcall that has no arguments and no return value. Make the body of the method simply print "Hello."

Solution.

```
public class Methodcall {
    public void sayHello( ) {
        System.out.println("Hello") ;
    }
}
```

7. Add another method start in the same class Methodcall. Make the start method of Methodcall call sayHello().

Solution.

```
public class Methodcall {
    :
    public void start( ) {
        sayHello( ) ;
    }
}
```

8. Add another method called addTwo that takes an integer argument, adds 2 to it, and returns that result.

Solution.

```
public class Methodcall {
    :
    public int addTwo(int i)
    {
        return i + 2 ;
    }
}
```

9. In the start method of Methodcall, define a local integer variable and initialize it to the result of calling addTwo(3). Print out the variable ; that is, print out the result of the method call. Define another local integer variable initialize it to the result of calling addTwo(19). Print out its result.

Solution.

```
public void start( ) {
    sayHello( );
    int a = addTwo(3) ;
    System.out.println("addTwo(3) is" +a) ;
    int b = addTwo(19) ;
    System.out.println("addTwo(19) is" +b) ;
}
```

10. Add method Test to Methodcall class and invoke start method from it.

Solution.

```
public void Test( ) {
    start( );
}
:
```

11. Why do you think function overloading must be a part of an Object Oriented Language ?

Solution. A function overloading must be a part of an Object Oriented Language as it is the feature that implements polymorphism in an object-oriented language. That is the ability of an object to behave differently in different circumstances can effectively be implemented in programming through function overloading.

Also with function overloading, the programmer is relieved from the burden of choosing the right function for a given set of values. This important responsibility is carried out by the compiler when a function is overloaded.

12. With the multiple definitions of single function name, what makes them significantly different ?

Solution. Even though there are multiple definitions with a single function name, these multiple definitions are distinguished by their signature. The argument list of a function is known as the signature of the function. Thus, even though the overloaded functions have similar names but they have different signatures which make them significantly different.

Even if the argument differs in terms of its signed-ness (signed vs. unsigned) and const-ness, the function is said to have different signature.

13. Illustrate the concept of function overloading with the help of an example.

Solution. A function name having several definitions that are differentiable by the number or types of their arguments, is known as function overloading.

For example, following code overloads a function area to compute areas of circle, rectangle and triangle.

```
float area (float radius) {      //computes area of a circle
    return 3.14 * radius * radius ;
}
float area (float length, float breadth)
{
    return length * breadth ;
}
float area (float side1, float side2, float side3) {
    float s = (side1 + side2 + side3) / 2 ;
    float ar = Math.sqrt (s * (s - side1) * (s - side2) * (s - side3)) ;
    return ar ;
}
```

14. Write a short note on significance of constructors in OOP classes.

Solution. One major characteristic of OOP is its close correspondence with the real-world entities. Like real-world entities, OOP objects also are created and scrapped. When an object is created, it must be *constructed* with some legal initial value automatically without being specified by the programmer. In this job of initializing an object, some legal value is carried out by the constructor of the class, the object belongs to. Therefore, the constructor takes over this very important duty of initialization of an object being created and relieves us from this task.

15. 'Only a function that has access to the constructor, can use the objects of this class'. Comment on this statement.

Solution. Since, every time an object is created, it is automatically initialized by the constructor of the class.

Therefore, it is very much necessary for the function using an object that it must have access to the constructor of that class so that the object being defined could be properly constructed. Thus, a function not having access to constructor of a class cannot declare and use objects of that class.

16. What do you mean by a temporary instance of a class ? What is its use ? How is it created ?

Solution. A temporary instance of a class means an anonymous object (object having no name) of the same class and which is shortlived. Its benefit is when an object is required only for very short time (say for an expression or a statement), we need not reserve memory for it for long. A temporary object for the same purpose can be created which remains in the memory as long as the statement defining it is getting executed, after the statement, this object is automatically destroyed and memory is released. Therefore, memory remains occupied only for the time when it is needed.

A temporary instance is created by explicit call to the constructor. For instance, following statement creates a temporary instance of Time type and invokes print() member function of Time class for it.

```
Time (10, 12, 30).print();
```

17. What will be the output of following ?

```
class MAIN {
    MAIN()
    {
        calculate();
        System.out.println ("constructor");
    }
    void calculate()
    {
        show();
        System.out.print ("calculating");
    }
    void show()
    {
        System.out.print ("I am displaying");
    }
    public static void main()
    {
        MAIN one = new MAIN();
    }
}
```

Solution. The output produced will be :

I am displaying calculating constructor

18. What is this keyword? What is its significance?

Solution. The member functions of every object have access to a sort of magic keyword named this, which points to the object itself. Thus any member function can find out the address of the object of which it is a member.

The this keyword represents an object that invokes a member function. It stores the address of the object that is invoking a member function and it (this keyword) is an implicit argument to the member function being invoked.

The this keyword is useful in returning the object (address) of which the function is a member.

19. What is recursion?

Solution. In a program, if a function calls itself (whether directly or indirectly), it is known as recursion. And the function calling itself is called recursive function e.g., following are two examples of recursion :

```
(i) void A( ) {  
    A( );  
}  
  
(ii) void B( ) {  
    C( );  
}  
void C( ) {  
    B( );  
}
```

20. Why is base case so important in a recursive function?

Solution. The base case, in a recursive case, represents a pre-known case whose solution is also preknown.

This case is very important because upon reaching at base case, the termination of recursive function executes endlessly. Therefore, the execution of base case is necessary for the termination of the recursive function.

21. Compare iteration and recursion.

Solution. In iteration, the code is executed repeatedly using the same memory space. That is, the memory space allocated once, is used for each pass of the loop.

On the other hand in recursion, since it involves function call at each step, fresh memory is allocated for each recursive call. For this reason i.e., because of function call overheads, the recursive function runs slower than its iterative counterpart.

22. Write a recursive function for printing fibonacci series up to 10 terms.

Solution. class fib {

```
int fibo ( int n ) {  
    if ( n == 1 )  
        return 0 ;  
    else if ( n == 2 )  
        return 1 ;  
    else if ( n > 2 )  
        return fibo(n - 1) + fibo (n - 2) ;  
    else  
        return - 1 ;  
}
```

```
public static void main( String args[ ] ) {  
    int i, term ;  
    fib obj1 = new fib( ) ;  
    for ( i = 1 ; i <= 10 ; ++i ) {  
        term = obj1.fibo( i ) ;  
        System.out.println( term ) ;  
    }  
}
```

glossary

Actual Parameters Also called arguments. The variables or values passed to a function.

Constructor A member function having the same name as its class and that initializes class objects with legal initial values.

Function A named unit of a group of program statements.

Function call Statement invoking a function

Function Declaration Also called function prototype. It is the statement that declares a function's name, return type, number and type of its arguments.

Function Declaration: First line of a function definition.

Function Definition A function declaration plus the statements in the body of the function.

Formal Parameters Also called parameters. The variables that receive their numeric values from the statements in the body of the function.

Function Prototype A function declaration.

Infinite Recursion: A recursive function executing endlessly.

this The keyword (which is actually a reference) storing the address of the object currently invoking a member function.

Temporary Object: An anonymous shortlived object.

Recursion Process of calling a function from within itself.

Recursive function: A function calling itself.

Assignments

TYPE A : VERY SHORT/SHORT ANSWER QUESTIONS

1. Function not returning any value has return type as :
(a) int (b) char (c) float (d) void.
 2. A function can return _____ values.
(a) 1 (b) 2 (c) 3 (d) all of the above.
 3. The parameters appearing in function call statement are called
(a) actual parameters (b) formal parameters
(c) call parameters (d) all of the above

12. Which of the following function-definitions are overloading the method given below :

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

- (a) `int sum(int x, int y, int z) { }`
- (c) `int sum (float x, float y) { }`
- (e) `float sum(int x, int y, float z) { }`

- (b) `float sum(int x, int y) { }`
- (d) `int sum (int a, int b) { }`

13. What is the principal reason for passing arguments by value ?

14. When an argument is passed by reference,

- (a) a variable is created in the function to hold the argument's value.
- (b) the function cannot access the argument's value.
- (c) a temporary variable is created in the calling program to hold the argument's value.
- (d) the function accesses the argument's original value in the calling program.

15. What is the role of a return statement in a function ?

16. What are the three types of functions in Java ?

17. What is that class called which does not have a public constructor ?

18. At what time is the constructor method automatically invoked ?

19. What is recursive function ?

20. Write one advantage of recursive functions.

21. What are the two cases required in a recursive function ?

22. What is base case ?

23. What is recursive case ?

24. Is it necessary to have a base case in a recursive function ? Why/Why not ?

TYPE B : SHORT ANSWER QUESTIONS

1. Define a function. What is function prototype ?

2. What are actual and formal parameters of a function ?

3. How many values can be returned from a function ?

4. Identify the errors in the function skeletons given below :

- (i) `float average (a, b) { }`
- (ii) `float mult (int x, y) { }`
- (iii) `float doer (int, float = 3.14) { }`

5. Given the function below write a `main()` function that includes everything necessary to call this function :

```
int thrice (int x)
{ return a * 3; }
```

6. What is the principal reason for passing arguments by value ? What is the principal reason for passing arguments by reference ? In a function call, what all data items can be passed by reference ?

7. Differentiate between CALL by reference and CALL by value.

8. What is polymorphism ? How does function overloading implement polymorphism ?

9. What is function overloading ? What is the significance of function overloading in Java ?

10. What is the role of a function's signature in disambiguation process ?

11. What is a constructor ? What does it do ?

12. Can you think of the benefits of a private class if any ? What are they ?

13. Write a class specifier (along with its constructor) that creates a class **student** having two private data members : rollno and grade and two public functions **init()** and **display()**.
 (Do not write full definitions of member functions except for constructor).
14. Here is a skeleton definition of a class :

```
class Sample {
    int i; char C; float f;
    :           // public members
}
```

Implement the constructor.

15. What condition(s) a function must specify in order to create objects of a class ?
16. Constructor functions obey the usual access rules. What does this statement mean ?
17. How are parameterized constructors different from non-parameterized constructors ?
18. What are benefits/drawbacks of temporary instances ?
19. How do we invoke a constructor ?
20. How can objects be initialized with desired values at the time of object creation ?
21. Write a function that takes two **char** arguments and returns 0 if both the arguments are equal. The function returns -1 if the first argument is smaller than the second and 1 if the second argument is smaller than the first.
22. Can each recursive function be represented through iteration ? Give examples.
23. Which of the following are examples of recursive functions :

```
(i) static void Print(char ch) {
    if(ch != ' ') {
        Print(ch);
        System.out.print(ch);
    }
}

(ii) static void recur(int p) {
    while(p - -)
        Recur(p);
    System.out.print("**");
}

(iii) static void recur(int p) {
    while(p - -)
        recur(p);
    System.out.print("#");
}

(iv) static void Check(int c) {
    Mate(c + 1);
}
void Mate(int d) {
    Check(d - 1);
}
```

```
(v) static void PrnNum(int n) {
    if(n == 1)
        return ;
    else {
        System.out.print(n) ;
        PrnNum(n - 2) ;
    }
}
```

24. Which of the functions shown above (in Q. 2) would result into infinite recursion ? Why ? What solution would you suggest for it ?

25. Identify the base case(s) in the following recursive function :

```
static public int fib(int n) {
    if(n == 1)
        return 0 ;
    else if(n == 2)
        return 1 ;
    else if(n > 0)
        return fib(n - 1) + fib(n - 2) ;
    else
        return -1 ;
```

26. The function shown in Q. 2(v) has been invoked as follows :

```
PrnNum(42) ;
```

But it is executing endlessly even though the function contains a base case. Find out the reason and correct it too.

27. Write a recursive function to calculate power of a number.

28. Write a recursive function that returns sum of squares of first 50 natural numbers.

OUTPUTS AND ERROR QUESTIONS

29. What is the output of the following code ?

```
void func(String s) {
    String s1 = s + "xyz" ;
    System.out.println("s1 =" + s1) ;
    System.out.println("s =" + s) ;
}
```

30. What is the output of the following program ?

```
class Static {
    static int m = 0 ;
    static int n = 0 ;
    public static void StaticTest() {
        int m = 10 ;
```

```

int x = 20 ;
int n = 30 ;
System.out.println("m + n =" + m + n) ;
}
x = m + n ;
System.out.println("x =" + x) ;
}
}

```

31. What is role of void keyword in declaring functions ?

TYPE C : LONG ANSWER QUESTIONS

- How is call-by-value method of function invoking different from call-by-reference method ? Give appropriate examples supporting your answer.
- Write a function that takes an int argument and doubles it. The function does not return a value.
- Write a function that takes two char arguments and returns 0 if both the arguments are equal. The function returns -1 if the first argument is smaller than the second and 1 if the second argument is smaller than the first.
- Write a complete Java program that invokes a function satis() to find whether four integers a, b, c, d sent to satis() satisfy the equation $a^3 + b^3 + c^3 = d^3$ or not. The function satis() returns 0 if the above equation is satisfied with the given four numbers otherwise it returns -1.
- Write a program uses a function power() to raise a number m to power n . The function takes int values for m and n and returns the result correctly. Use a default value of 2 for n to make the function calculate squares when this argument is omitted. Write a main() to get the value of m and n to display the calculated result.
- How does the compiler interpret more than one definitions having same name ? What steps does it follow to distinguish these ?
- Discuss how the best match is found when a call to an overloaded function is encountered. Give example(s) to support your answer.
- When a compiler can automatically generate a constructor if it is not defined then why is it considered that writing constructors for a class is a good practice ?
- 'Accessibility of a constructor or a destructor greatly affects the scope and visibility of their class.' Elaborate this statement.
- List some of the special properties of the constructor functions.
- What is a parameterized constructor ? How is it useful ?
- Design a class to represent a bank account. Include the following members :

Data members

 - Name of the depositor
 - Type of account
 - Account number
 - 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
 - Do write proper constructor functions.

Program Error Types and Basic Exception Handling

9.1 Introduction

Learning a programming language involves lot of things such as learning the basic syntax, programming concepts, compiling and running code, and how to rectify errors. When you write a program, you may encounter different errors and different types of errors. Thus, it is important to learn about various types of errors and how to handle them. Java supports a specific and well-defined mechanism of catching and preventing errors of any type that may occur. This mechanism is known as *Exception Handling*. In this chapter you are going to learn about exception handling techniques, different types of errors that may occur and ways to avoid them.

This chapter is dedicated to the discussion of the same. It first talks about different types of program errors and ways of handling them, and then it talks about a related concept – Exceptions and Exception Handling in Java.

In This Chapter

- 9.1 Introduction
- 9.2 Type of Program Errors
- 9.3 Exception and Exception Handling
- 9.4 Exception Hierarchy
- 9.5 Forcing an Exception
- 9.6 Benefits of Exception Handling

9.2 Type of Program Errors

An error, sometimes called 'a bug', is anything in the code that prevents a program from compiling and running correctly. Some program bugs are catastrophic in their effects, while others are innocuous and still others are so obscure that you will never discover them. There are broadly three types of errors : *Compile-time errors*, *run-time errors* and *logical errors*.

9.2.1 Compile-Time Errors

Errors that occur during compilation, are compile-time errors. When a program compiles, its source code is checked for whether it follows the programming language's rules or not. Two types of errors fall into category of compile-time errors : *syntax errors* and *semantics errors*.

1. Syntax errors

Syntax errors occur when rules of a programming language are misused i.e., when a grammatical rule of Java is violated.

For example, observe the following two statements :

```
X - Y * Z ;
if X = (X * Y) ...
```

These two statements will result in syntax errors as two operators cannot come in continuation in an expression, and '=' is the assignment operator, not a relational operator. Therefore, the correct statements will be as follows :

```
X = Y * Z ;
if (X == (X * Y))
```

Some other syntax errors in Java are like :

Example 1 : String not enclosed in double quotes

```
String str = "This is a long string' ;
```

Error : String literal is not properly closed double-quotes.

Example 2 : Extra parenthesis in if expression

```
if (i > j) )
    max = i;
```

Error : Syntax error on token ")", delete this token.

Example 3 : Missing right-hand side

```
max = ;
```

Error : Expression expected after token '='.

Compile-Time Errors

Syntax errors

Semantics

Syntax refers to formal rules governing the construction of valid statements in a language.

Example 4 : Keyword is used as the name for variable

```
int default = 10;
```

Error : Invalid variable name.

Example 5 : Missing return value

```
int max(int i, int j) {  
    return; // Error, return type is int, it must return an int value  
}
```

Error : This method must return a result of type int.

One should always try to get the syntax right the first time, as a syntax error wastes computing time and money, as well as programmer's time, and it is preventable.

2. Semantics Errors

Semantics Errors occur when statements are not meaningful.

For instance, the statement 'Sita plays Guitar' is syntactically and semantically correct as it has some meaning but the statement 'Guitar plays Sita' is syntactically correct (as the grammar is correct) but semantically incorrect. Similarly, there are semantics rules of a programming language, violation of which results in semantical errors.

Semantics refers to the set of rules which give the meaning of a statement.

For instance, the statement

```
X * Y = Z ;
```

will result in a semantical error as an expression cannot come on the left side of an assignment operator.

Example 1 : Use of a non-initialized variable :

```
int i;  
i++;
```

The variable *i* is not initialized

Example 2 : Type incompatibility :

```
int a = "hello";
```

The types String and int are not compatible.

Example 3 : Errors in expressions :

```
String s = "...";  
int a = 5 - s;
```

The - operator does not support arguments of type String.

Handling Compile Time Errors

All syntax errors and some of the semantic errors (the static semantic errors) are detected by the compiler, which generates a message indicating the type of error and the position in the Java source file where the error occurred (notice that the actual error could have occurred even before the position signaled by the compiler).

To correct a compile time error, you should do this :

1. Look carefully at the position/line number indicated by the compiler for the error mentioned.
2. Then look carefully at the code at the given line number. If you find any syntax/semantic violation, correct that.
3. If however, you do not find any problem in the code at the given line number, look before i.e., in the lines earlier than the given line number for the actual reason. e.g., In the following code, the compiler reported error at line 6 :

```

1. void m(int n) { // n is initialized from the actual parameter
2.     int i, j;
3.     i = 2;           // i is initialized by the assignment
4.     int k = 1;       // k is initialized in its declaration
5.     if (i == n)      // OK
6.         k = j;        // Error, j is not initialized
7.     else
8.         j = k;
9. }

Error : Line 6 : Variable j might not have been initialized.

```

But the code at line 6 is syntactically accurate. Thus, you must look at the lines above line 6, i.e., in lines 5, 4, 3, 2. And you will find that the code's error is because of line 2 where variable *j* could have been initialized. So you can correct the line 2 as :

```
int i, j = 3;
```

After this correction, if you recompile your code, it will be free from compile time errors.

4. Correct the code where you sense the real problem lies (e.g., in above example code, it was line 2 and not line 6 that caused the problem).
5. Recompile the code.
6. Repeat until your code is error free.

9.2.2 Run-Time Errors

Errors that occur during the execution of a program are run-time errors. These are harder to detect errors. Some run-time errors stop the execution of the program which is then called program "crashed" or "abnormally terminated".

Most run-time errors are easy to identify because program halts when it encounters them e.g., an infinite loop or wrong value (of different data type other than required) is input.

If your program crashes, Java will print the stack trace. The stack trace is a list of all of the methods that were being executed when the program crashed. You'll find that the bug is almost always in one of those methods.

Note A **run-time error** is an error that occurs during the execution of a program. In contrast, **compile-time errors** occur while a program is being compiled.

Some common run-time errors are like :

Example 1 : Division by zero :

```
int a, b, x;
a = 10;

b = Integer.parseInt(kb.readLine());
x = a/b;           //ERROR if b = 0
```

"Division by Zero" error will occur only if user has entered value 0 for variable **b**.

Example 2 : File does not exist :

```
FileReader f = new FileReader("names.txt");
```

This error occurs only if the file **names.txt** does not exist on the disk.

Example 3 : Dereferencing of a null reference :

```
String s, t;
s = null;
t = s.concat("a");
```

Note that the above code is syntactically correct, but it contains a dynamic semantic error due to the fact that a method cannot be applied to a string reference whose value is **null** and string **s** is **null**, hence the error.

Handling Run-Time Errors

By default, Java terminates your program displaying the stack trace if any run-time error occurs. But a better way of handling these errors is through Exception Handling, which are covered in details in sections 9.3 onwards.

9.2.3 Logical Errors

Sometimes, even if you don't encounter any error during compile-time and run-time, your program does not provide the correct result. This is because of the programmer's mistaken analysis of the problem he or she is trying to solve. Such errors are *logical errors*. For instance, an incorrectly implemented algorithm, or use of a variable before its initialization, or unmarked end for a loop, or wrong parameters passed are often the hardest to prevent and to locate. These must be handled carefully. Sometimes logical errors are treated as a subcategory of run-time errors.

A **logic error** is a mistake that complies with the rules of the compiler that causes the program to generate incorrect output.

Some examples of logical errors are like :

Example 1 : Errors in the performed computation :

```
// specification : find the sum of two given numbers
public static int sum(int a, int b) {
    return a - b; // returning difference instead of sum
}
```

Logically this code is wrong as this method returns the wrong value w.r.t. the specification that requires the sum of two integers.

Example 2 : Non-termination :

```
String s = br.readLine();
while (s != null) {
    System.out.println(s);
}
```

This loop will never terminate ever as the string `s` will never be `null` as long as the loop is running ; and to stop the loop, the condition is that the string `s` should be `null`, which will never be true and the loop will never terminate.

Handling Logical Errors

Logical errors are not straight forward, thus require a different mechanism. Most common approaches to handle logical errors are:

1. Desk-checking and Dry Run

It refers to following the code/algorithm line by line and checking how it affects the variables and their values. It is unplugged activity i.e., it is performed using pen and paper, e.g., (see below)

```
void test()
{
    1. int number = 3;
    2. System.out.println(number);
    3. for(int i = 1 ; i<=3; i++) {
        4.     number = number + 5 ;
        5.     System.out.println(number) ;
    6. }
    System.out.println("?");
}
```

Line	number	i	Output
1	3		
2			
3		1	
4	8		
5			
3		2	8
4	13		
5			
3		3	13
4	18		
5			
6			18
			?

2. Inserting Tracing Statements

This method requires you to insert temporary display statements to display intermediate calculations and result. This is done when :

- ❖ When program "crashes".
- ❖ Program runs completely but produces incorrect output.

For example, (see below) the code on the left is original code and code on the right having the tracing statements inserted into.

Tracing Statements

```
void chk(int n) {
    int i = 2, j = 2, k=3;
    int p, q, r ;
    for ( k = 1 ; k <10; k ++)
        for ( j = 10-k ; j > 0; j--) {
            p = n + i *j + j + k;
            q = p + j*k;
            r = q-p;
            n = n/2;
        }
    System.out.println(q);
}
```

```
void chk(int n) {
    int i = 2, j = 2, k=3;
    int p, q, r ;
    for ( k = 1 ; k <10; k ++)
        for ( j = 10-k ; j > 0; j--) {
            System.out.println(j + " " + k);
            p = n + i *j + j + k;
            q = p + j*k;
            System.out.println(p + " " + q);
            r = q-p;
            n = n/2;
            System.out.println(r + " " + n);
        }
    System.out.println(q);
}
```

With this method, you get to know where the problem is occurring when you look at the displayed intermediate values. This way you can pinpoint and correct the error. However, after correcting the error, you should remove the tracing statements from the code.

3. Using an Interactive Debugger

Most compilers provide interactive debugger with them. You can run your code with it. It will execute your code line by line and will show its impact on the variables. It can simultaneously show the line of code executing and the corresponding values of variables.

9.3 Exception and Exception Handling

Exception in general refers to some contradictory or unexpected situation or in short an error that is unexpected. During program development, there may be some cases where the programmer does not have the certainty that this code-fragment is going to work right, either because it accesses to resources that do not exist or because it gets out of an unexpected range, etc.... These types of anomalous situations are generally called *exceptions* and the way to handle them is called *exception handling*.

Contradictory or Unexpected situation or unexpected error, during program execution, is known as **Exception**.

You already know that broadly there are two types of errors :

- (i) **Compile-time errors.** These are the errors resulting out of violation of programming language's grammar rules e.g., writing syntactically incorrect statement like

```
System.out.println "A Test" ;
```

will result into compile-type error because of invalid syntax. All syntax errors are reported during compilation.

- (ii) **Run-time errors.** The errors that occur during runtime because of unexpected situations. Such errors are handled through exception handling routines of Java.

Exception handling is a transparent and nice way to handle errors that occur during program run.

Many reasons support the use of exception handling. In other words, advantages of exception handling are :

- (i) Exception handling separates error-handling code from normal code.
- (ii) It clarifies the code (by removing error-handling code from main line of program) and enhances readability.
- (iii) It stimulates consequences as the error-handling takes place at one place and in one manner.
- (iv) It makes for clear, robust, fault-tolerant programs.

Way of handling anomalous situations in a program-run, is known as **Exception Handling**.

So we can summarize Exception as :

It is an exceptional event that occurs during runtime and causes normal program flow to be disrupted.

Some common examples of Exception are :

- | | |
|-------------------------------|---|
| ◆ Divide by zero errors | ◆ Accessing the elements of an array beyond its range |
| ◆ Invalid input | ◆ Hard disk crash |
| ◆ Opening a non-existent file | ◆ Heap memory exhausted |

For instance, consider the following code :

```

1  class DivByZero {
2      public static void main(String args[]) {
3          System.out.println( 3/0 ); This will cause Divide by Zero exception
4          System.out.println("Pls. print me.");
5      }
6  }
```

If execute above code, you'll receive a message as :

```

Exception in thread "main"
java.lang.ArithmaticException: / by zero
at DivByZero.main(DivByZero.java:3)
```

This message is generated by *default exception handler* of Java. The default exception handler is provided by Java runtime that does the following upon occurrence of an Exception :

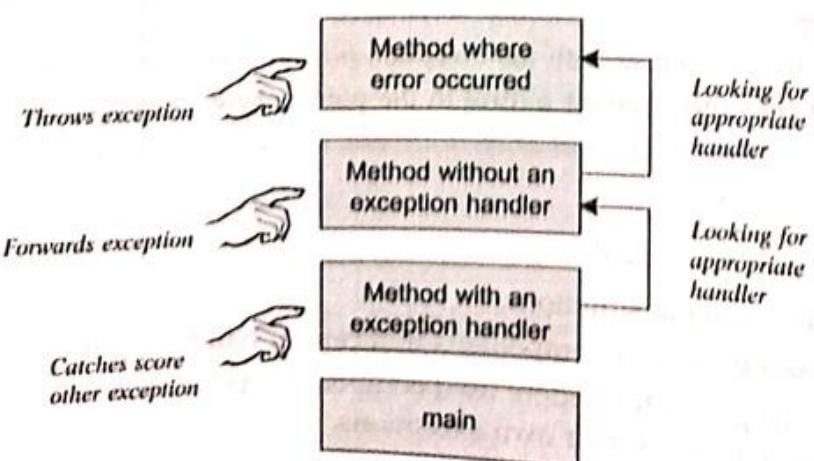
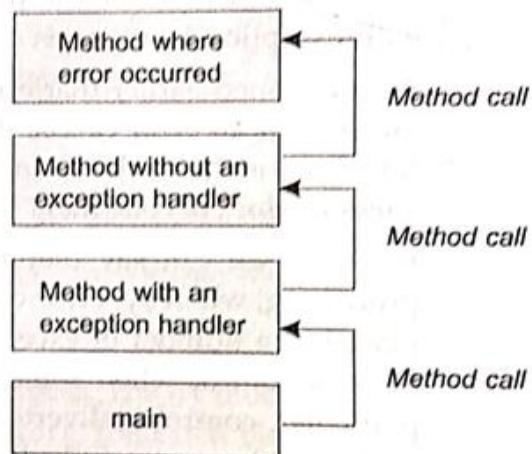
- (i) Prints out exception description
- (ii) Prints the *stack trace*, i.e., ◆ hierarchy of methods where the exception occurred
- (iii) Causes the program to terminate

Let us learn how it all happens ?

9.3.1 What happens when an Exception occurs ?

When an exception occurs within a method, some activities take place internally. These can be summarized in the following way :

1. The method creates an *exception object* and hands it off to the *runtime system*. Creating an exception object and handing it to the runtime system is called **throwing an exception**. The created *exception object* contains information about the *error*, including its type and the state of the program when the error occurred.
2. The runtime system searches the call stack for a method that contains an exception handler. (see adjacent figure).
3. When an appropriate handler is found, the runtime system passes the exception to the handler
 - ◆ An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler
 - ◆ The exception handler chosen is said to catch the exception.
4. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates and uses the default exception handler.



9.3.2 Concept of Exception Handling

The global concept of error-handling is pretty simple. That is, write your code in such a way that it raises some error flag everytime something goes wrong. Then trap this error flag and if this is spotted, call the error handling routine. The intended program flow should be somewhat like the one shown in Fig. 9.1.

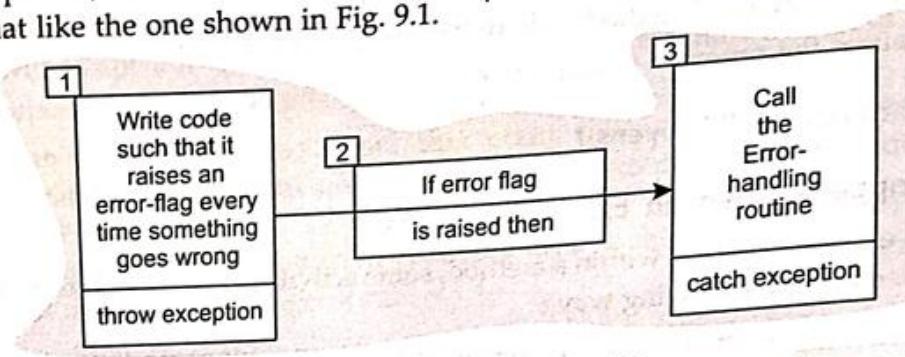


Fig. 9.1 Concept of exception handling.

The *raising* of imaginary error flag is called *throwing* an error. When an error is thrown, the overall system responds by *catching* the error. And surrounding a block of error-sensitive-code-with-exception-handling is called *trying* to execute a block.

As mentioned earlier that error-handling takes place at one place and in one manner. This means that an error can be thrown over function boundaries. That is, if one of the deepest functions on the stack has an error, this error can propagate up to the upper function if there is a *trying-block* of code there. This property helps one to keep error-handling code at one place.

An exception can be generated by an error in the JVM, an error in your program, or explicitly via a *throw* statement.

In short, we can say that Exception handling supports the notion of generalized error processing, whereby, error correction code can be separated from the main body of code and placed in a number of exception handlers. In other words, exceptions are for changing the flow of control when something important or unexpected, normally an error, occurs. In particular, control is diverted to another part of the program that can try to cope with the error, or at least gracefully die'. Some terminology used within exception handling follows:

Description	Java Terminology
An unexpected error that occurs during runtime.	Exception
The process by which an exception is generated and passed to the program.	Throwing
Capturing an exception that has just occurred and executing statements that try to resolve the problem.	Catching
The block of code that attempts to deal with the exception (<i>i.e.</i> , problem).	Catch clause, or catch block
The sequence of method calls that brought control to the point where the exception occurred.	Stack trace

When to Use Exception Handling

The exception handling is ideal for :

- ❖ processing exceptional situations.
- ❖ processing exceptions for components that cannot handle them directly.
- ❖ processing exceptions for widely used components (such as libraries, classes, functions) that should not process their own exceptions.
- ❖ large projects that require uniform error-processing.

9.3.3 Exception Handling in Java

Java treats *exceptions* as objects that describe any error caused by an external resource not being available or an internal processing problem. They (the Exception type objects) are passed to exception handlers written by the programmer to enable graceful recovery. If the handler has not been written, the program will terminate with a display of the Exception class. There are many exception classes such as *IOException* (error while handling IO) and *NumberFormatException* (error when invalid format of number is encountered) and a general exception class *Exception* that can catch virtually all exceptions.

Using try{} and catch{}

To intercept, and thereby control, an exception, you use a *try/catch/finally* construction. You place lines of code that are part of the normal processing sequence in a *try* block. You then put code to deal with an exception that might arise during execution of the *try* block in a *catch* block. If there are multiple exception classes that might arise in the *try* block, then several *catch* blocks are allowed to handle them.

The following piece of code illustrates a simple exception handler.

```
try
{
    file = new DataInputStream( new InputStream( fileName ) );
}
catch( Exception e )
{
    System.out.println( "\n Exception occurred" );
}
```

This is the code where a runtime error or exception may occur, hence it is enclosed in try block.

This is the block that will determine what to do if the exception of mentioned type occurs during execution.

An exception handler consists of two core sections, called blocks. The *try* block encloses some code which might throw an exception (*i.e.*, generate an error). The *catch* block contains the error handling code, *i.e.*, determines what should be done if an error is detected. Hence, exception handling provides a means of separating error handling from any code that might result in errors. In many situations this is beneficial as it produces 'cleaner' code.

Handling Exceptions

Java uses the *try-catch-finally* syntax to test (*i.e.*, try) a section of code and if an error occurs in that region, to trap (*i.e.*, catch) the error. Any number of catches can be set up for various exception types. The *catch{ }* block is also responsible for handling the exceptions. The *finally* keyword can be used to provide a block of code that is always performed regardless of whether an exception is signaled or not.

The syntax for using this functionality is :

```
try {
    // tested statement(s);
}
```

If the code in try block encounters an error, exception is thrown.

Note Code that must be executed no matter what happens can be placed in a *finally* block.

```

    catch(ExceptionName e1)
    {
        // trap handler statement(s) ;
    }
    catch (ExceptionName e2) // any number of catch statements
    {
        // trap handler statement(s) ;
    }
    finally  This block will always be
    {                                         executed in the last
        // always executed block
    }

```

catch blocks trap the thrown exception and handle them

Note A catch block can accept one exception only. For multiple exceptions multiple catch blocks are to be written.

Forcing an Exception

The **throw** keyword (note the singular form) is used to force an exception. That means, you as programmer can force an exception to occur through **throw** keyword. It can also pass a custom message to your exception handling module. For example :

```
throw new FileNotFoundException("Could not find result.txt") ;
```

Now consider the following code in which exception occurs during runtime. Since, this code does not provide any error-handler, Java's default error-handler handles it.

Note The point at which **throw** is executed is termed as **throw point**.

Program 9.1

Program that can generate exception but has no error-handling routine.

```

1 public class DivideException {
2     public static void main(String[] args) {
3         division(100,4);      // Line 1
4         division(100,0);      // Line 2
5         System.out.println("Exit main().");
6     }
7
8     public static void division(int totalSum, int totalNumber) {
9         System.out.println("Computing Division.");
10        int average = totalSum/totalNumber;
11        System.out.println("Average : "+ average);
12    }

```

The output produced is

```

Computing Division.
java.lang.ArithmaticException: / by zero
Computing Division.
at DivideException.division(DivideException.java:9)
at DivideException.main(DivideException.java:4)
Exception in thread "main"

```

When run, an **ArithmaticException** is thrown at runtime when Line 9 is executed because integer division by 0 is an illegal operation. The "Exit main()" message is never reached in the main method.

Now if you write your own error handling routine using try-catch block in the program 9.1, your program changes to similar to the one given in program 9.2.

Program 9.2

Program that can generate exception and has error-handling routine.

```
public class DivideException2 {
    public static void main(String[] args) {
        int result = division(100,0); // Line 2
        System.out.println("result : "+ result);
    }
    public static int division(int totalSum, int totalNumber) {
        int quotient = -1;
        System.out.println("Computing Division.");
        try{
            quotient = totalSum/totalNumber; This may cause Divide  
By Zero exception.
        }
        catch(Exception e){
            System.out.println("Exception : "+ e.getMessage());
        }
        finally{
            if(quotient != -1){
                System.out.println("Finally Block Executes");
                System.out.println("Result : "+ quotient);
            } else {
                System.out.println("Finally Block Executes. Exception Occurred");
                return quotient;
            }
        } The output produced is
        return quotient;
    }
}
```

*Computing Division.
Exception : / by zero
Finally Block Executes. Exception Occurred
result : -1*

Rules for *try*, *catch* and *finally* Blocks

While writing error-handling routines in your programs, you must follow the following rules.

1. For each *try* block there can be zero or more *catch* blocks, but only one *finally* block.
2. The *catch* blocks and *finally* block must always appear in conjunction with a *try* block.
3. A *try* block must be followed by either at least one *catch* block or one *finally* block.
4. The order exception handlers in the *catch* block must be from the most specific exception.

The *finally* Clause of Exception Handling

Now that you have learnt to work about *finally* clause, let us talk about some other important things of *finally* clause. You know that whether or not an Exception is thrown, a *finally* clause can be provided for performing local clean-up and releasing non-object resources etc.

The JVM follows following rules for the finally clause of Exception Handling.

- ◆ A *try* block cannot be exited without executing the corresponding *finally* clause.
- ◆ The code in a *finally* clause is executed whether or not an exception is caught.
- ◆ If a transfer of control flow occurs in the *try* block because of a *break*, *continue* or *return* statement, then the *finally* clause is executed before flow transfer occurs.

Following program illustrates this.

Program 9.3

Program to illustrate handling of finally clause in Exception Handling routine.

```

import java.io.*;
public class TryCatchBreak {
    public static void main( String[] args ) {
        BufferedReader in = null ;
        String lineIn ;
        for ( int i = 0 ; i < 5 ; i++ ) {
            try {
                in = new BufferedReader( new FileReader( "poem.txt" ) );
                if ( i < 4 ) continue ; ← // continue for loop
                while ((lineIn = in.readLine()) != null )
                    System.out.println(lineIn);
            }
            catch (Exception e) { ← A try block cannot be exited without
                System.out.println("Exception occurred.");
            }
            finally {
                System.out.println("Value in loop is: " + i );
                try {
                    in.close() ; // close file
                }
                catch (Exception e2) { }
            } // end of finally
        } // end for loop
    } // end main
}

```

These lines are printed because finally block is executed before exiting from try block.

Again this line is printed when finally block is executed while exiting from try block.

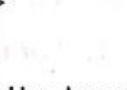
Value in loop is: 0 *Value in loop is: 1* *Value in loop is: 2* *Value in loop is: 3*

*If school were more like baseball
we'd only have to play.
We'd hang out in the sunshine
and run around all day.*

*We wouldn't have to study.
We'd practice and we'd train.
And, best of all, they'd cancel
whenever there was rain.*

Value in loop is: 4




The throws Clause

If a method invokes code that causes an exception, then the method should provide a catch clause to handle the exception. If a catch does not handle the exception, then the exception must be passed out of the method and be handled by the code calling the method. If an exception is allowed to pass through a method, a **throws** clause is required in the method declaration to indicate that an exception can occur that the method itself does not handle. It is specified with method header, e.g.,

```
public static void main(String[ ] args) throws Exception {  
    :  
}
```

Indicating that if an exception occurs, it will automatically report it to error handler or processor.

If a method has no **throws** clause, then the method cannot pass or throw any exceptions. Multiple Exception types (comma separated) can be declared in the **throws** clause. A method must declare in its **throws** clause all the checked¹ exceptions it can throw e.g.,

```
public static void main(String[ ] args) throws IOException, NumberFormatException  
{ ... }
```

When accessing a method that can throw an Exception, *three* approaches are possible for handling the Exception :

1. Catch the Exception and handle it.
2. Catch the Exception, map the Exception to a new user-defined Exception type, and throw the new Exception type. The new user-defined Exception must be declared in the method statement.
3. Declare the Exception in the **throws** clause of the current method and let the Exception pass through the method.

The general rule is to catch those Exceptions that you know how to handle and propagate those Exceptions that you do not know how to handle.

9.3.4 The Exception Class

The **catch** statement also stores an instance of the exception that was caught in the variable that the programmer uses, in the previous example *Exception e*. While all exceptions are subclasses of *Exception*, *Exception* itself is a subclass of *Throwable*, which contains a nice suite of methods that you can use to get all kinds of information to report about your exceptions :

- ◆ *getMessage()* : returns the error message reported by the exception in a *String*
- ◆ *printStackTrace()* : prints the stack trace of the exception to standard output, useful for debugging purposes in locating where the exception occurred
- ◆ *printStackTrace(PrintStream s)* : prints the stack trace to an alternative output stream
- ◆ *printStackTrace(PrintWriter s)* : prints the stack trace to a file, this way you can log stack traces transparent to the user or log for later reference
- ◆ *toString()* : if you just decide to print out the exception it will print out this:
NAME_OF_EXCEPTION: getMessage().

1. Checked exception will become clear in a later section.

9.3.5 Common Exceptions

There are many different exceptions that can be thrown by a program, and the Java API contains quite a few. A lot are contained in the default package, *java.lang*; however, when you start using more functionality such as AWT, Swing, or *java.io*, the packages may also contain additional exceptions thrown by those libraries. As you start expanding the functionality, it might be a good idea to look at potential exceptions in the package and when they might be thrown in the course of your application. Here is a primer of some :

- ◆ *ArithmaticException*—thrown if a program attempts to perform **division by zero**
- ◆ *ArrayIndexOutOfBoundsException*—thrown if a program attempts to access an index of an array that does not exist
- ◆ *StringIndexOutOfBoundsException*—thrown if a program attempts to access a character at a non-existent index in a *String*
- ◆ *NullPointerException*—thrown if the JVM attempts to perform an operation on an *Object* that points to no data, or *null*
- ◆ *NumberFormatException*—thrown if a program is attempting to convert a string to a numerical datatype, and the string contains inappropriate characters (*i.e.*, 'z' or 'Q')
- ◆ *ClassNotFoundException*—thrown if a program cannot find a class it depends at runtime (*i.e.*, the class's ".class" file cannot be found or was removed from the CLASSPATH)
- ◆ *IOException*—actually contained in *java.io*, but it is thrown if the JVM failed to open an I/O stream

Now have a look at following program that traps and handles a *NumberFormatException*.

Program 9.4

Program to trap and handle NumberFormatException (when undesired and invalid number format is encountered).

```

import java.io.* ;
public class Program13_4 {
    static boolean valid ;
    private static BufferedReader stdin = new BufferedReader
        (new InputStreamReader(System.in)) ;
    public static void main(String[ ] args) throws IOException
    {
        int n = 0;           See, try { } block surrounds the statement which may cause exception.
        try {               Here it is z = x / 4 ; as it may cause divide-by-zero exception.
            System.out.print("Enter an integer: ") ;
            //Reading below line of text from the user.
            int input = Integer.parseInt(stdin.readLine()) ;
            valid = true ;   //this statement only executes if the string
                            //entered does not cause a NumberFormatException
        }
        catch(NumberFormatException e) {
            //A NumberFormatException occurred, print an error message
            System.out.println(e.getMessage() + " is not a valid format for an integer.") ;
        }
    }
}

```

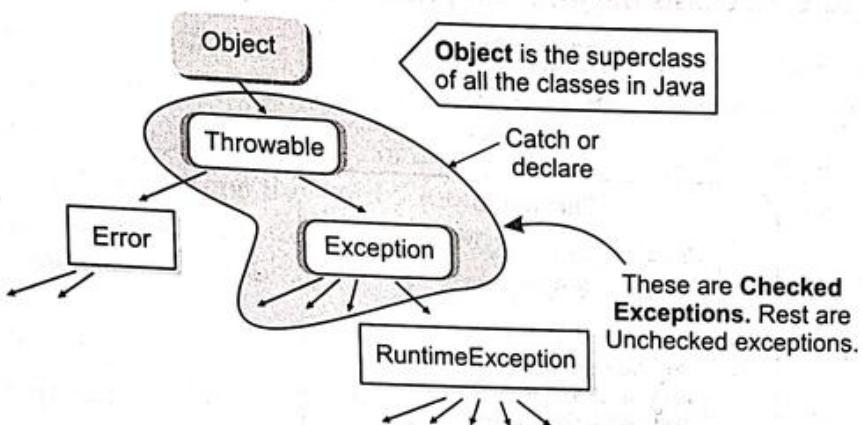
The output produced is

Enter an integer: xyz
For input string: "xyz" is not a valid format for an integer.

Catch block immediately follows the try block with the exception type it handles

9.4 Exception Hierarchy

Below is a simplified diagram of the exception hierarchy. The *Throwable* class is the superclass of all Errors and Exceptions. An *Error* object describes internal errors, for example, resource exhaustion, inside the Java run-time system. An *Exception* object describes a recoverable error that should be handled by the programmer.



Let us talk about this class hierarchy :

Throwable class

It is the root class of exception classes. Its immediate subclasses are :

- ◆ *Error* class
- ◆ *Exception* class

Exception class

This class handles conditions that user programs can reasonably deal with. The exceptions are usually the result of some flaws in the user program code.

Some examples of Exceptions are:

- ◆ *Division by zero error*
- ◆ *Array out-of-bounds error*

Error class

This class is used by the Java run-time system to handle errors occurring in the run-time environment, which are generally beyond the control of user programs. Some examples of Errors are :

- ◆ *Out of memory errors*
- ◆ *Hard disk crash*

What is the difference between an Error and an Exception ?

An *Error* indicates that a non-recoverable error has occurred that should not be caught. Errors usually cause the Java Virtual Machine to display a message and exit.

According to Sun, the makers of Java :

"An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions."

For example, one of the subclasses of *Error* is named *VirtualMachineError*. This error is "Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating."

On the other hand, an **Exception** indicates an abnormal condition that must be properly handled to prevent program termination. *Sun* explains it this way :

"The class **Exception** and its subclasses are a form of **Throwable** that indicates conditions that a reasonable application might want to catch."

Following figure 9.2 enlists the Java Exception Hierarchy :

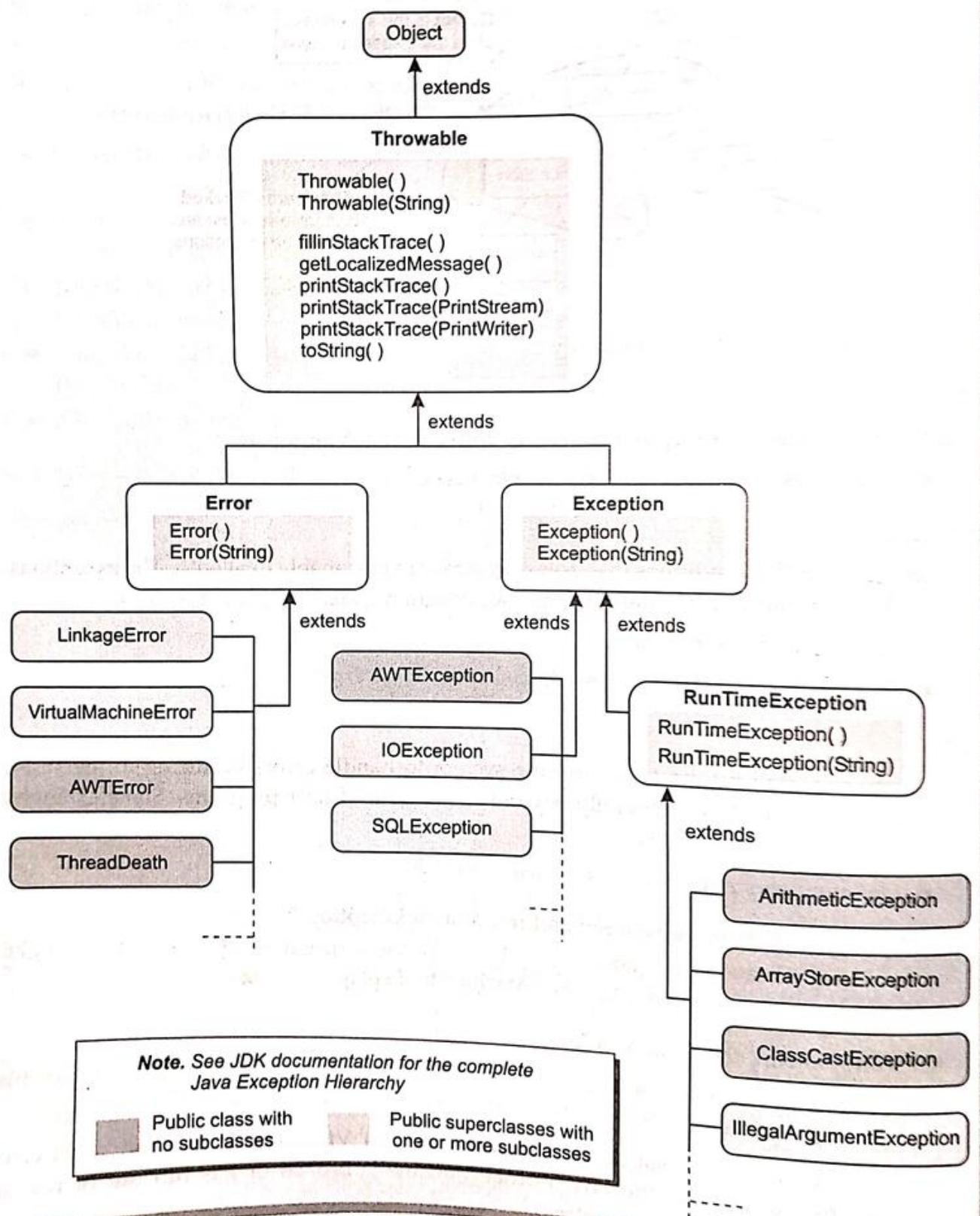


Figure 9.2 Java exception hierarchy.

Checked vs Unchecked Exceptions

The Exceptions in Java belong to one of the following *two* categories :

- ❖ Checked exception
- ❖ Unchecked exceptions

Checked exceptions

These represent invalid conditions in areas outside the immediate control of the program (invalid user input, database problems, network outages, absent files).

Checked Exceptions are subclasses of *Exception* class. A method is *obliged* to establish a policy for all checked exceptions thrown by its implementation (either pass the checked exception further up the stack, or handle it somehow). If the program neither catches nor lists the occurring checked exception, compiler error will occur.

Unchecked exceptions

These exceptions are not subject to compile-time checking for exception handling. These represent defects in the program (bugs) — often invalid arguments passed to a non-private method. To quote from *The Java Programming Language*, by Gosling, Arnold, and Holmes :

"Unchecked run-time exceptions represent conditions that, generally speaking, reflect errors in your program's logic and cannot be reasonably recovered from at run-time."

The built-in unchecked exception classes are :

- ❖ Error
- ❖ RuntimeException
- ❖ Their subclasses

A method is *not* obliged to establish a policy for the unchecked exceptions thrown by its implementation (and they almost always do not do so). Handling all these exceptions may make the program cluttered and may become a nuisance.

9.5 Forcing an Exception

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the *throw* keyword. A *throw* statement causes the current method to immediately stop executing, much like a return statement, and the exception is thrown to the previous method on the call stack. For example, the following statement throws a new *ArrayIndexOutOfBoundsException*, with 5 being the invalid index :

```
throw new ArrayIndexOutOfBoundsException(5);
```

You can also instantiate an exception object and then throw it in a separate statement :

```
ArrayIndexOutOfBoundsException a = new ArrayIndexOutOfBoundsException(5);
//Some time later
throw a;
```

```

Or    try {
        Exception e1 = new Exception("User defined exception") ;
        //Some time later
        throw e1 ;
    }
    catch(Exception e) {
        System.out.println("ERROR: " + e.getMessage());
    }

```

You can throw only objects of type `java.lang.Throwable`. In almost all situations, you will throw an object that is a child of `java.lang.Exception`. Recall that the `Exception` class extends the `Throwable` class.

The `throw` keyword is used to force an exception. Ordinarily a new `Exception` object is created, then it is *thrown*. The act of throwing an exception causes the execution of the code within the method to be immediately terminated and control is transferred to the "closest" exception handler. "Closest" here means the method that is closest in the chain of method calls. For example, if method `main()` calls method `A()` which calls method `B()` which finally calls method `C()`, the call chain looks like

`main() → A() → B() → C()`

If method `C()` throws an exception and method `A()` has the code to handle that exception, `A()` is the closest method up the call chain that can handle the exception. In this case when `C()` throws the exception :

1. `C()`'s execution is interrupted by the `throw`.
2. Normally control would return from method `C()` back to method `B()` (this is the normal method call return path), but the `throw` statement causes the return to `B()` to be bypassed.
3. Any code following the call to `B()` within `A()`'s normal program flow is ignored and the exception handling code within `A()` is immediately executed.

9.6 Benefits of Exception Handling

Exception provides the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In short, the advantages of exception handling are :

1. It separates the working/functional code from the error-handling code by way of `try-catch` clauses.
2. It allows a clean path for error propagation. If the called method encounters a situation it can't manage, it can throw an exception and let the calling method deal with it.
3. By enlisting the compiler to ensure that "exceptional" situations are anticipated and accounted for, it enforces powerful coding.
4. It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of `try-catch` blocks.

With this we have reached the end of this chapter.

Creating user Defined Exceptions

Though Java provides an extensive set of in-built exceptions, there are cases in which we may need to define our own exceptions in order to handle the various application specific errors that we might encounter.

While defining a user defined exception, we need to take care of the following aspects :

- ii The user defined exception class should extend from *Exception* class.
- ii The *toString()* method may be overridden in the user defined exception class in order to display meaningful information about the exception.

Let us see a simple example to learn how to define and make use of user defined exceptions.

NegativeAgeException.java

```
public class NegativeAgeException extends Exception {
    private int age;
    public NegativeAgeException(int age) {
        this.age = age;
    }
    public String toString() {
        return "Age cannot be negative" + " " + age ;
    }
}
```

CustomExceptionTest.java

```
public class CustomExceptionTest {
    public static void main(String[] args) throws Exception {
        int age = getAge();
        if (age < 0) {
            throw new NegativeAgeException(age);
        } else {
            System.out.println("Age entered is " + age);
        }
    }
    static int getAge() {
        return -10;
    }
}
```

In the **CustomExceptionTest** class, the age is expected to be a positive number. It would throw the user defined exception **NegativeAgeException** if the age is assigned a negative number.

At runtime, we get the following exception since the age is a negative number.

Exception in thread "main" Age cannot be negative -10

at tips.basics.exception.CustomExceptionTest.main(CustomExceptionTest.java:10)

L et us Revise

- ❖ Broadly the program errors can either be compile time errors or run-time errors.
- ❖ Compile time errors can be syntax errors or semantic errors.
- ❖ Run-time errors can be logical errors or dynamic semantic errors.
- ❖ Way of handling anomalous situations in a program run is known as exception handling.
- ❖ The throws keyword automatically informs the error occurrence so that it can be trapped.
- ❖ The try block is for enclosing the code wherein exceptions can take place.
- ❖ The catch block traps the exception and handles it.
- ❖ The throw keyword forces an exception.
- ❖ All exceptions are subclasses of `Throwable class`.
- ❖ Exceptions belong to one of the two categories: *Checked and Unchecked exceptions*.
- ❖ The Checked exceptions must be handled by a proper error handler routine.

S olved Problems

1. What are the types of program errors?

Solution. There are mainly these types of program errors :

- (i) **Syntax errors.** Errors that occur when program writing rules (the syntax) of a language are violated.
- (ii) **Semantic errors.** Errors that occur when the code statements follow the syntax but are improperly formed.
- (iii) **Logical errors.** Errors that occur when the given specification for the program is not followed and the program is running but not producing the correct result.
- (iv) **Run-time errors.** Errors that occur during run-time mostly because of unavailability of some result or some unexpected values.

2. What is an Exception ?

Solution. Exception in general refers to some contradictory or unusual situation which can be encountered while executing a program.

3. When is Exception Handling required ?

Solution. The exception handling is ideal for :

- ☒ processing exceptional situations.
- ☒ processing exceptions for components that cannot handle them directly.
- ☒ processing exceptions for widely used components that should not process their own exceptions.
- ☒ large projects that require uniform error-processing.

4. What is the function of catch block in exception handling ? Where does it appear in a program ?

Solution. A catch block is a group of Java statements that are used to handle a raised exception. The catch blocks should be placed after each try block.

5. What are the advantages of exception handling ?

Solution. Advantages of exception handling are :

- (i) Exception handling separates error-handling code from normal code.

- (ii) It clarifies the code and enhances readability.
- (iii) It stimulates consequences as the error-handling takes place at one place and in one manner.
- (iv) It makes for clear, robust, fault-tolerant programs.

6. When do you need multiple catch handlers ?

Solution. Sometimes program has more than one condition to throw exceptions. In such cases, one can associate more than one catch statement with a try.

7. What is wrong with this fragment ?

```
// ...
    vals[18] = 10 ;
    catch (ArrayIndexOutOfBoundsException exc) {
        // handle error
    }
```

Solution. There is no try block preceding the catch statement.

8. The finally block is the last bit of code executed before your program ends. True or False? Explain your answer.

Solution. False. The finally block is the code executed when a try block ends or is exited.

9. What type of exceptions must be explicitly declared in a throws clause of a method?

Solution. All exceptions except those of type RuntimeException and Error must be declared in a throws clause.

10. What are the two direct subclasses of Throwable ?

Solution. Error and Exception.

11. What are the three ways that an exception can be generated ?

Solution. An exception can be generated by an error in the JVM, an error in your program, or explicitly via a throw statement.

12. What happens when you run a program that creates an array of ints and then sets the value of an array component whose index is greater than the length of the array ?

Solution. The following code tries to set component 60 in an array of length 50 :

```
public class sp11 {
    public static void main(String[] args) {
        int[] ints = new int[50];
        ints[60] = 12345;
    }
}
```

The code compiles but throws an exception when executed. The exact message may vary, but the following is typical :

java.lang.ArrayIndexOutOfBoundsException at sp11.main(sp11.java:4) Exception in thread "main".

13. What is the purpose of the finally clause of a try-catch-finally statement ?

Solution. The finally clause is used to provide the capability to execute code no matter whether or not an exception is thrown or caught.

14. What classes of exceptions may be caught by a catch clause ?

Solution. A catch clause can catch any exception that may be assigned to the Throwable type. This includes the Error and Exception types.

15. Can an exception be rethrown ?

Solution. Yes, an exception can be rethrown. And it will be rethrown to the caller method to handle the exception. This process continues till a method that handles this exception is called.

16. What class of exceptions are generated by the Java run-time system ?

Solution. The Java runtime system generates RuntimeException and Error exceptions.

17. What happens if a try-catch-finally statement does not have a catch clause to handle an exception that is thrown within the body of the try statement ?

Solution. The exception propagates up to the next higher level try-catch statement (if any) or results in the program's termination.

18. How does a try statement determine which catch clause should be used to handle an exception ?

Solution. When an exception is thrown within the body of a try statement, the catch clauses of the try statement are examined in the order in which they appear. The first catch clause that is capable of handling the exception is executed. The remaining catch clauses are ignored.

19. What is the difference between Exception and Error ?

Solution. The Exception class defines mild error conditions that our program encounters. Exceptions can occur when trying to open the file, which does not exist, the network connection is disrupted, operands being manipulated are out of prescribed ranges, the class file you are interested in loading is missing.

The Error class defines serious error conditions that we should not attempt to recover from. In most cases, it lets the program terminate when such an error is encountered.

20. What is the difference between throw and throws clause ?

Or

Differentiate between throw and throws with respect to exception handling.

Solution. Keyword **throw** is used to throw an exception manually, whereas keyword **throws** is used in the case of checked exceptions, to tell the compiler that we haven't handled the exception, so that the exception will be handled by the calling function.

21. What would be printed by the following code segment if someValue equals 1000 ?

```
int M = someValue ;
try {
    System.out.println("Entering try block") ;
    if (M > 100)
        throw new Exception(M + " is too large") ;
    System.out.println("Exiting try block") ;
}
catch (Exception e) {
    System.out.println("ERROR : " + e.getMessage()) ;
}
```

Solution. If someValue equals 1000, the code segment will print :

```
Entering try block
ERROR : 1000 is too large
```

Glossary

Exception An anomalous situation encountered by the program.

Syntax error Programming language's grammar rules violation error.

Compile time error Error that the compiler can find during compilation.

Run-time error Error during program execution.

Accessor Methods Methods used to obtain information about an object.

Java package Group of related classes and interfaces.

Assignments

1. What are program errors ? What are the types of program error ?
2. How are compile time errors different from run-time errors ?
3. What are the types of compile time errors ?
4. What are logical errors ? Why are they hard to find ?
5. How will you handle compile time errors ?
6. How will you handle logical errors ?
7. Name the block that encloses the code that may encounter anomalous situations.
8. In which block can you throw the exception ?
9. Which block traps and handles exception ?
10. Can one catch block sufficiently trap and handle multiple exceptions ?
11. Which block is always executed no matter which exception is thrown ?
12. NumberFormatException is subclass of which exception class ?
13. IOException is subclass of which exception class ?
14. What is the role of (i) throws keyword ? (ii) throw keyword ?
15. Why is exception handling necessary ?
16. How do you throw an exception ?
17. How do you handle an exception ?
18. Describe the keyword try. How it is used ?
19. What is the root class for all exceptions ?
20. What type of exceptions require the caller to explicitly handle them ?
21. What type of exceptions do not require the caller to explicitly handle them ?
22. What is the throws keyword, and how does it differ from the throw keyword ?
23. Which of the code fragments will throw an "ArrayOutOfBoundsException" ?
 - (a)

```
for (int i = 0 ; i < args.length ; i ++ ) {  
    System.out.print( i );  
}
```

- (b) System.out.print(args.length);
- (c) for (int i = 0 ; i < 1; i++) {
 System.out.print(args[i]);
 }
- (d) None of the above

24. Write a try/catch block that throws an Exception if the value of variable X is less than zero. The exception should be an instance of Exception, and, when it is caught, the message returned by getMessage() should be "ERROR : Negative value in X coordinate".

25. What line of a given program will throw FileNotFoundException ?

```
import java.io.*;
public class MyReader {
    public static void main ( String args[ ] ) {
        try {
            FileReader fileReader = new FileReader("MyFile.java");
            BufferedReader bufferedReader = new BufferedReader(fileReader);
            String strString;
            fileReader.close();
            while ( ( strString = bufferedReader.readLine() ) != null ) {
                System.out.println ( strString );
            }
        }
        catch ( IOException ie) {
            System.out.println ( ie.getMessage() );
        }
    }
}
```

26. 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.

27. What is a finally block ? When and how is it used ? Give a suitable example.

10.1 Introduction

In the previous chapter, you have learnt to create and use classes. Apart from the classes created by you, you can also use classes provided by Java.

To help programmers be more productive, Java includes predefined classes in the form of *packages* as part of the installation. These are called the *Java class libraries*. Java offers many packages through its libraries. Packages, in turn, provide thousands of classes that provide tens of thousands of methods for carrying out diverse type of tasks. Availability of these useful classes makes life much easier for Java programmers. Some of the most used packages are : *language extensions java.lang, utilities java.util, input-output (io) java.io, GUI java.awt and javax.swing, applets java.applet, network services java.net, etc.*

In this chapter we are going to learn to use some of the *java.lang* and *java.io* classes basically for input/output purposes and for using strings. We shall also be talking about wrapper classes briefly.

Firstly we shall learn how to obtain input from user and provide output. Although you've been introduced to the syntax to obtain input in a previous chapter, yet this chapter is going to talk about the concepts behind the same. While reading input, certain types of unexpected errors may occur. To trap and counter these errors, we shall also be learning about exception handling in Java.

In This Chapter

- 10.1 Introduction
- 10.2 Simple Input/Output
- 10.3 Exception Handling
- 10.4 Wrapper Classes
- 10.5 Working with Strings
- 10.6 Packages in Java

10.2 Simple Input/Output

Input is any information that is needed by your program to complete its execution. There are many forms that program input may take, such as *character input*, *mouse click*, *audio input*, *graphic input* or *file input* etc. etc. **Output** is any information that the program must convey to the user. The information you see on your computer screen is being output by one or more programs that are currently running on your computer. When you decide to print a document, a program is told to send some output to the printer. Any sound that your computer makes is because some program sent output to the speakers on your computer. The possibilities for program output are also limited (*just like input*) only by our imaginations.

Although Java is a powerful language yet it offers no standard statements for obtaining input or for providing output. In Java, all input and output takes places using methods that are part of `java.io` package. In order to make available all the classes inside a package, you need to write :

```
import <packagename>.*  
e.g., import java.io.*
```

will make available all classes and their methods in `java.io` package.

Simple Output

Let me tell you an interesting thing – you have been already told how to send output to the monitor. Right from the very first program that you wrote – the `HelloWorld` application, you have been using `System.out.println` for displaying text on monitor. Well, `System.out.println` is a way to send output to the monitor. Let us recall our `HelloWorld` program.

```
class HelloWorld {  
    public static void main (String[ ] args) {  
        System.out.println("Hello World !") ;  
    }  
}
```

`System.out` refers to an output stream managed by the `System` class that implements the standard output stream. `System.out` is an instance of the `PrintStream` class defined in the `java.io` package. The `PrintStream` class is an `OutputStream` that is easy to use. Simply call one of the `print`, `println`, or `write` methods to write various types of data to the stream.

The difference between `print` and `println` is that after `print` statement, next successive output takes place at the same line whereas with `println`, the next successive output takes place at the next line. The reason being that `println` automatically adds *newline character* (`\n`) at the end of the text being printed. Consider following code fragment that illustrates the difference between the two functions :

```
class HelloWorld {  
    public static void main (String[ ] args) {  
        System.out.println("Println at work") ;  
        System.out.println("Hello World !") ;  
        System.out.println("Thank You") ;  
    }  
}
```

```

        System.out.println("____");
        System.out.println("Now Print at work"); // This statement is printed with a println
        System.out.print("Hello World ! ");
        System.out.print("Thank You");
    }
}

```

The output produced will be as follows :

Println at work

Hello World !

Thank You

See the text by two printin's have been printed in two lines because the printin had appended newline character at the end of the text being printed. Hence successive output takes place at next line.

Now Print at work

Hello World ! Thank You

*See the text by two prints have been printed in the same lines because no newline character was * appended at the end of the text being printed. Hence successive output takes place at the same line.*

Now can you guess the output of following lines ?

```

System.out.println("Hello World !");
System.out.print ("Thank You");
System.out.println("____");
System.out.print("Now Print at work");
System.out.println("Hello World !");
System.out.print("Thank You");

```

You guessed it right. It indeed is :

Hello world !

Thank You

Now Print at work Hello World !

Thank You

The write function is used to print selective bytes from array of bytes. Note that the bytes will be written as given ; to write characters that will be translated according to the platform's default character encoding, use the **print(char)** or **println(char)** methods. We are not discussing write here to keep our discussion simple.

Simple Input

To read characters from the standard input device, generally our very own keyboard, we can use **System.in.read** to read in characters entered at the keyboard by the user.

Carefully read the following code fragment that illustrates the usage of **System.in.read** function :

```

class Count {
    public static void main (String[ ] args)
        throws java.io.IOException {
            int oneInt = 0;
}

```

You'll learn about exceptions and exception handling in a later section of this chapter.

```

    oneInt = System.in.read();
    Function that reads one character from
    keyboard and returns as integer i.e., equivalent ASCII or Unicode value.

    System.out.println("You entered" + oneInt);
}

}

```

System.in refers to an input stream managed by the **System** class that implements the standard input stream. **System.in** is an *InputStream* object. *InputStream* is an abstract class¹ defined in the *java.io* package that defines the behavior of all sequential input streams in Java. All the input streams defined in the *java.io* package are subclasses of *InputStream*. *InputStream* defines a programming interface for input streams that includes methods for reading from the stream, marking a location within the stream, skipping to a mark, and closing the stream.

Streams ! ☺ What are streams ? Let us find out.

IO Streams

In Java, a source of input data is called an **input stream** and the output data is called an **output stream**. Think of these streams like the ones shown in Fig. 10.1.

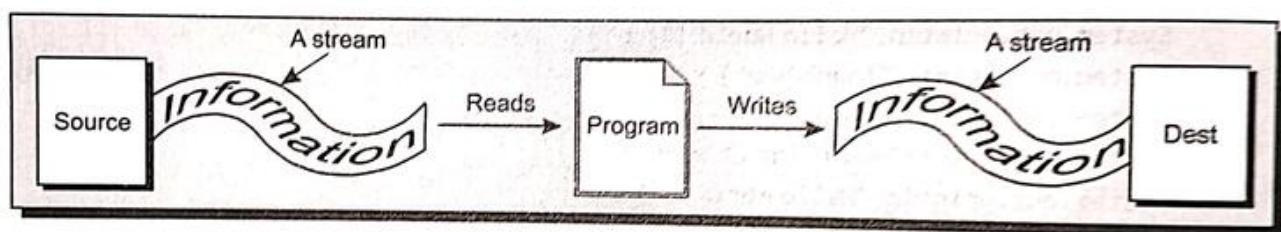


Figure 10.1 Streams

In this picture, each character in the stream is supposed to be a datum (chunk of data) waiting in line to be input, or leaving as output. Inputting data is usually called *reading* data ; and outputting data is usually called *writing* data (or *printing* data if the output stream is connected to a monitor or a printer.)

Your Java program inputs one character at a time from the input stream and outputs one character at a time to output stream.

Your program may have several input streams flowing into it and several output streams flowing out of it. Mostly following three IO streams are used in Java programs :

- ◆ **System.in** – the input stream.
- ◆ **System.out** – the output stream for normal results.
- ◆ **System.err** – the output stream for error messages.

Normally **System.in** is connected to the *keyboard* (the *standard input device*) and the *data are characters*. **System.out** and **System.err** both are connected to the *monitor* (the *standard output device* and the *standard error device respectively*), and also are *character data*. These streams are difficult to manage by themselves. But we need not worry, Java is always readily available for help through the package *java.io*.

1. An abstract class is the one whose objects cannot be created, however, other classes can inherit from it.

Streams can be one of following *two types* :

- (i) **ByteOriented.** Where there is no notion of datatype or encoding scheme such as Unicode. Thus we can read data of any datatype with them as bytes are read. These streams are known as data streams – *data input streams* and *data output streams*.
- (ii) **CharacterOriented.** Where a character encoding scheme is used. Obviously only characters are read through these types of streams. These streams are often referred to as *Reader* and *Writer* streams.

For byte-oriented IO (or binary IO) Data Streams are used and for character based I/O Readers and Writers are used. Let us first learn to work with *CharacterOriented IO*.

Make sure to import java.io package in your program or import the desired class by specifying the class e.g., import.java.io. ; will import all the classes in java.io package but import.java.io.DataInputStream ; will import only the DataInputStream class of java.io package.*

Character Oriented IO

Let us now seriously learn to obtain character based console input through methods other than that `System.in.read()` i.e., through *Readers* and *Writers*. *Readers* and *Writers* are like other input and output streams. The only difference is that they are exclusive oriented to *Unicode characters*.

There are many ways to get information from the user. In many cases, the user must be told that they should enter some information. This is known as *prompting the user*.

When a program is waiting for input at the console, there is sometimes a blinking cursor in the console window indicating that the user should type some information. But, this is not always the case. The user will only know what information to type, if the program describes that information in the form of a user prompt.

The use of several of the Java I/O classes is required to successfully receive input that is typed by the user. Although there are 50+ classes in `java.io` package, we just need to work with three of them to successfully retrieve the input. All three classes are in the `java.io` package. To work with them, you need to either use the fully qualified name shown or import the `java.io` package.

- ◆ `java.io.InputStream` stores information about the connection between an input device and the computer or program.
- ◆ `java.io.InputStreamReader` translates data bytes received from `InputStream` objects into a stream of characters.
- ◆ `java.io.BufferedReader` buffers (stores) the input received from an `InputStreamReader` object. This is done to improve the efficiency. Input devices are much slower than the computer's processor and buffering the data reduces the number of times the CPU has to interact with the device itself.

We will create an instance of the `InputStreamReader` class to be used to create the `BufferedReader` object that we want. What do we need to have in order to create an instance of the `InputStreamReader` class ? According to the Java API, we will need an `InputStream`

object. Now you must have understood why we needed all three classes ! Luckily, part of our work is done for us. The **System** class in the **java.lang** package automatically creates an **InputStream** object that is connected to the keyboard. It is called **System.in** and is part of the **java.lang** package.

We will use the **System.in** object to create an instance of the **InputStreamReader** class and then use that object to create an instance of the **BufferedReader** class. For obtaining console based user input following steps should be followed.

Steps for console based user input :

1. Use the **System.in** object to create an **InputStreamReader** object.

```
// Creating an InputStreamReader using the standard input stream.  
InputStreamReader isr = new InputStreamReader(System.in) ;
```

2. Use the **InputStreamReader** object to create a **BufferedReader** object.

```
// Creating a BufferedReader using the InputStreamReader.  
BufferedReader stdin = new BufferedReader(isr) ;
```

3. Display a prompt to the user for the desired data.

```
// Now prompting the user  
System.out.print("Type some data for the program :") ;
```

4. Use the **BufferedReader** object to read a line of text from the user using **readLine** function that reads a lines of characters typed by the user.

```
// Using the BufferedReader to read a line of text from the user.  
String input = stdin.readLine() ;
```

5. Do as required with the input received from the user.

```
// Now, you can do anything with the input string that you need to.  
// e.g., output it to the user.  
System.out.println("input = " + input) ;
```

You can also combine steps 1 & 2 and create only one instance of the **BufferedReader** for use throughout their entire program e.g.,

```
/* Combining steps 1 & 2. Creating a single shared BufferedReader for  
keyboard input */  
private static BufferedReader stdin =  
    new BufferedReader(new InputStreamReader(System.in)) ;
```

All keyboard operations can now use this single shared **BufferedReader** object (named **stdin** here). The code given above should be placed with other class data members and not inside any method.

Here's a complete program example that prompts the user for input and then repeats that data to the console window.

Note The **System.in** is an object of **InputStream** type and is automatically created and connected to the keyboard, by the **System** class. The **System.in** object is part of standard **java.lang** package.

Program 10.1

Obtain some input from the user and print it on the monitor.

```

import java.io.*; //needed for BufferedReader, InputStreamReader, etc.

/** A Java program that demonstrates console based input and output. */
public class MyConsoleIO {
    //Firstly creating a single shared BufferedReader for keyboard input
    private static BufferedReader stdin
        = new BufferedReader(new InputStreamReader (System.in) );
    //Program execution starts here
    public static void main(String[ ] args) throws IOException
    {
        //Promting the user
        System.out.print("Type some data for the program: ");
        String input = stdin.readLine(); //Reading a line of text from the user.
        System.out.println("input = " + input); //Displaying the input back to the user.
    } // main method ends here
} // MyConsoleIO class ends here

```

Write either this line with function's header or use try { }, catch {} blocks while inputting data (as shown in prog 10.2).

The output produced by above program is as follows :

Type some data for the program: Javs is fun
input = Java is fun

Integer Input

See, isn't it easy getting data from the user ? But wait, can you use this data in calculations – even if the user entered digits such as 9 or 5 ? Well even if the user types digits e.g., "123", that will be still be returned as a **String** object by the **readLine** method of **BufferedReader**. You will need to **parse** (convert) the **String** object into an **int** value if you wish to store it in an **int** variable or data member. Let us learn how you can do this :

1. Get a **String** of characters that is in an integer format, e.g., "123".

```
String input = stdin.readLine(); // from console input example above.
```

2. Use the **Integer** class to parse the string of characters into an integer.

This input is the string
which was read in
step1.

```
int number = Integer.parseInt (input);
// converts a String into an int value
```

Note The **Integer** wrapper class contains conversion methods for changing **String** data into **int** values and vice versa. The **Integer** class is one of several wrapper classes that are defined in the standard Java API. Wrapper classes have class methods for parsing and are also used when you need to store a primitive value as an object.

In our case, we needed to convert a **String** object into an **int** value. The **parseInt** method of the **Integer** class performs this action. Be sure to read the topic **Wrapper Classes** covered in a later section of this chapter for getting more information about this and other methods of the **Integer** class.

What if the user types letters instead of digits ?

Now if the user types letters instead of digits, the **parseInt** method declares that it may throw a **NumberFormatException**. If the user types any string of characters that can't be parsed into an **int** value, a **NumberFormatException** will be thrown and the program will crash. That can't be a good thing ! But, there is something that you as the programmer can do to keep your program from crashing. You can *catch* the **NumberFormatException**. Section 10.3 talks about exception handling in details.

Byte Oriented IO

As mentioned earlier, **DataStreams** are used for byte oriented IO or binary IO. Two **DataStreams** used for IO are :

- (i) **DataOutputStream** for providing output.
- (ii) **DataInputStream** for obtaining input.

Here we are concerned about taking input keyboard. Let us briefly talk about these classes.

A **DataInputStream** lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a **DataOutputStream** to write data that can later be read by a data input stream.

Methods from the classes **DataInputStream** and **DataOutputStream** read and write binary data. In general you cannot display these files with utilities such as **cat** (UNIX) and **type** (DOS, Windows) or **Notepad** (Windows).

Here are some of the methods
from **DataOutputStream** :

```
void writeBoolean(boolean v) ;
void writeChar(int v) ;
void writeInt(int v) ;
void writeDouble(double v) ;
void writeFloat(float v) ;
void writeInt(int v) ;
void writeLong(long v) ;
void writeShort(int v) ;
```

Here are some of the methods
from **DataInputStream** :

```
void readBoolean(boolean v) ;
void readChar(int v) ;
void readDouble(double v) ;
void readFloat(float v) ;
void readInt(int v) ;
void readLong(long v) ;
void readShort(int v) ;
```

But while using these classes for console IO, make sure that **DataInputStream** type class is associated with standard input i.e., **System.in** e.g.,

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

Similarly, make sure that **DataOutputStream** type class is associated with standard output i.e., **System.out** e.g.,

```
DataOutputStream output = new DataOutputStream(System.out) ;
```

In the following lines we are going to discuss **DataInputStream**, which you can use to obtain input from keyboard. For printing output, you need not use **DataOutputStream** class as you can directly send output to standard output device i.e., monitor using **System.out.println** or **System.out.print** method.

After instantiating a **DataInputStream** object (say **in**), you can read data as follows :

```
int n = in.readInt( ) ;
i = in.readInt( ) ;           / assuming that i is an already declared integer variable.
char c = in.readChar( ) ;
ch = in.readChar( ) ;         / assuming that ch is an already declared character variable
double d = in.readDouble( ) ;
:
```

Java makes exception handling mandatory. Therefore, you are supposed to enclose an input reading statement in a **try** block as depicted below. Immediately below **try** block, a **catch** block should follow :

```
try {
    // input reading statement here
}
catch(Exception e){ }
```

For now, just follow the above guidelines. Exception handling is being covered in details in Section 10.3. Following program (program 10.2) is reading an integer from the user and determines whether it is an even number or odd.

Program 10.2

Read an integer using **DataInputStream** and print whether it is odd or even.

```
import java.io.*;
public class Program14_2 {
    public static void main(String[ ] args) {
        DataInputStream in = new DataInputStream(System.in) ;
        int n = 0 ;
        try {
            System.out.println("Enter a digit") ;
            n = in.readInt( ) ;
        }
        catch(Exception e) {
            if(n % 2 == 0)
                System.out.println("You entered an even number. ") ;
            else
                System.out.println("You entered an odd number. ") ;
        }
    }
}
```

See, here data has been input through try {} and catch {} blocks.

The output produced is

Enter a digit

9

You entered an odd number

10.3 Exception Handling

Exception in general refers to some contradictory or unexpected situation or in short an error that is unexpected. During program development, there may be some cases where the programmer does not have the certainty that this code-fragment is going to work right, either because it accesses to resources that do not exist or because it gets out of an unexpected range, etc.... These types of anomalous situations are generally called exceptions and the way to handle them is called *exception handling*.

Contradictory or Unexpected situation or unexpected error, during program execution, is known as **Exception**.

Broadly there are *two* types of errors :

(i) **Compile-time errors.** These are the errors resulting out of violation of programming language's grammar rules e.g., writing syntactically incorrect statement like

```
System.out.println "A Test" ;
```

will result into compile-type error because of invalid syntax. All syntax errors are reported during compilation.

Way of handling anomalous situations in a program-run, is known as **Exception Handling**.

(ii) **Run-time errors.** The errors that occur during runtime because of unexpected situations. Such errors are handled through exception handling routines of Java.

Exception handling is a transparent and nice way to handle program errors.

Many reasons support the use of exception handling. In other words, advantages of exception handling are :

- (i) Exception handling separates error-handling code from normal code.
- (ii) It clarifies the code (by removing error-handling code from main line of program) and enhances readability.
- (iii) It stimulates consequences as the error-handling takes place at one place and in one manner.
- (iv) It makes for clear, robust, fault-tolerant programs.

10.3.1 Concept of Exception Handling

The global concept of error-handling is pretty simple. That is, write your code in such a way that it raises some error flag everytime something goes wrong. Then trap this error flag and if this is spotted, call the error handling routine. The intended program flow should be somewhat like the one shown in Fig. 10.2.

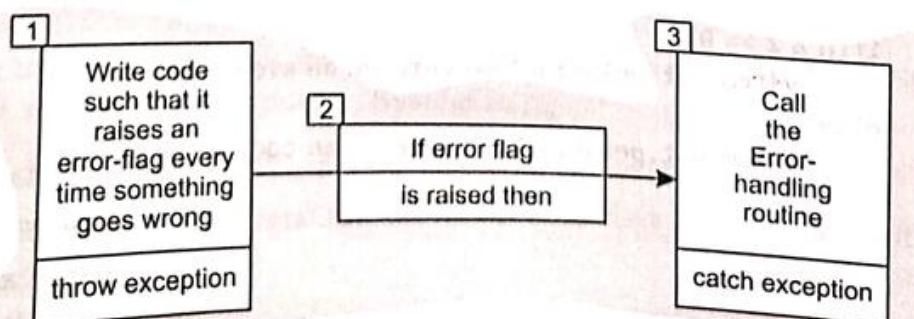


Figure 10.2 Concept of exception handling.

The *raising* of imaginary error flag is called *throwing* an error. When an error is thrown, the overall system responds by *catching* the error. And surrounding a block of error-sensitive-code-with-exception-handling is called *trying* to execute a block.

As mentioned earlier that error-handling takes place at one place and in one manner. This means that an error can be thrown over function boundaries. That is, if one of the deepest functions on the stack has an error, this error can propagate up to the upper function if there is a *trying-block* of code there. This property helps one to keep error-handling code at one place.

When to Use Exception Handling

The exception handling is ideal for :

- ❖ processing exceptional situations.
- ❖ processing exceptions for components that cannot handle them directly.
- ❖ processing exceptions for widely used components (such as libraries, classes, functions) that should not process their own exceptions.
- ❖ large projects that require uniform error-processing.

10.3.2 Exception Handling in Java

Java treats *exceptions* as objects that describe any error caused by an external resource not being available or an internal processing problem. They (the Exception type objects) are passed to exception handlers written by the programmer to enable graceful recovery. If the handler has not been written, the program will terminate with a display of the Exception class. There are many exception classes such as *IOException* (error while handling IO) and *NumberFormatException* (error when invalid format of number is encountered) and a general exception class *Exception* that can catch virtually all exceptions.

Using try{ } and catch{ }

To intercept, and thereby control, an exception, you use a *try/catch/finally* construction. You place lines of code that are part of the normal processing sequence in a *try* block. You then put code to deal with an exception that might arise during execution of the *try* block in a *catch* block. If there are multiple exception classes that might arise in the *try* block, then several *catch* blocks are allowed to handle them. Code that must be executed no matter what happens can be placed in a *finally* block.

The throws Keyword

Keyword *throws* is used to inform that an error has occurred. It is specified with method prototype e.g.,

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



indicating that if an IO error occurs, it will automatically report it to error handler or processor.

The Java compiler is aware of how some methods may cause specific exceptions and it forces you to deal with these immediately. If you choose not to write a handler, you can use the *throws xxxException* clause on the surrounding class to abdicate responsibility. (Refer to the

code example under **Simple Input** subsection of Section 10.2). For example, using `System.input()` will automatically give a compile error for `IOException` if `throws IOException` is written in the function prototype.

Handling Exceptions

Java uses the *try-catch-finally* syntax to test (*i.e.*, try) a section of code and if an error occurs in that region, to trap (*i.e.*, catch) the error. Any number of catches can be set up for various *exception types*. The `catch{ }` block is also responsible for handling the exceptions. The *finally* keyword can be used to provide a block of code that is **always performed** regardless of whether an exception is signaled or not.

The syntax for using this functionality is :

```
try {  
    // tested statement(s);  
}  
catch(ExceptionName e1)  
{  
    // trap handler statement(s);  
}  
catch (ExceptionName e2) // any number of catch statements  
{  
    // trap handler statement(s);  
}  
finally {  
    // always executed block  
}
```

Note A catch block can accept one exception only. For multiple exceptions multiple catch blocks are to be written.

Forcing an Exception

The `throw` keyword (note the singular form) is used to force an exception. That means, you as programmer can force an exception to occur through `throw` keyword. It can also pass a custom message to your exception handling module. For example,

```
throw new FileNotFoundException("Could not find result.txt");
```

Some basic I/O related exception classes and their functions are given below in Table 10.1.

Table 10.1 Basic I/O Exception Classes and Their Functions

I/O Exception class	Function
<code>EOFException</code>	Signals that an end of the file or end of stream has been reached unexpectedly during input.
<code>FileNotFoundException</code>	Informs that a file could not be found.
<code>InterruptedIOException</code>	Warns that an I/O operation has been interrupted.
<code>IOException</code>	Signals that an I/O exception of some sort has occurred.

Using Hierarchy Effectively

The exception classes are in a hierarchy (see Fig. 10.3). Catches for specific exceptions should always be written prior to the generic handler. For example, since *FileNotFoundException* is a child of *IOException*, a catch for *FileNotFoundException* should occur before the one for *IOException*. The latter handler will catch those exceptions that are missed by individual child handlers. And a generic handler for *Exception* would cover any missing situation e.g.,

```
try {
    :
}
catch(FileNotFoundException e1) { // exception handler that'll catch all FileNotFoundException exceptions
    :
}
catch(IOException e2) {      // exception handler that'll catch all IO exceptions
    :
}
catch(Exception e3) {       // generic exception handler that'll catch all other exceptions
    :
}
```

Now consider the following example program (Program 10.3) that handles a *NumberFormatException*.

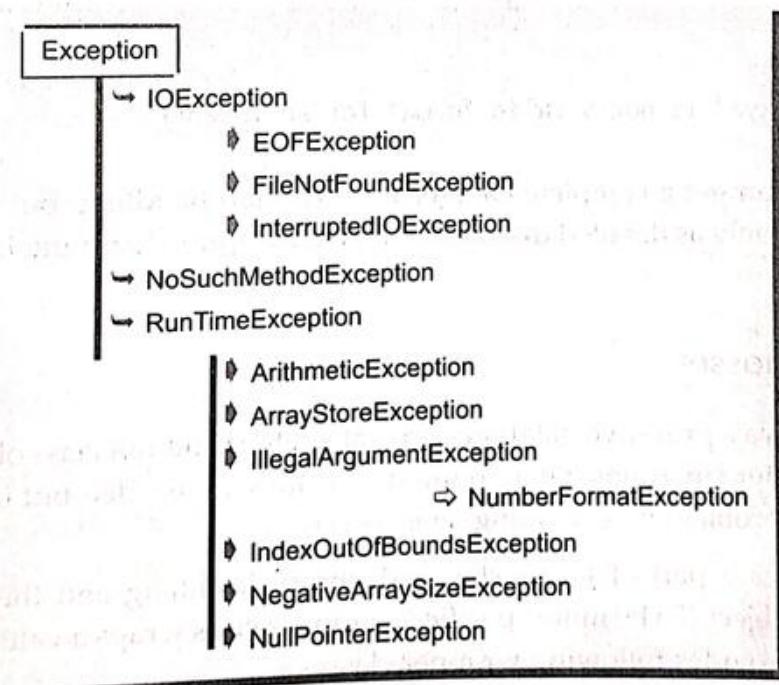


Figure 10.3 Exception Hierarchy of Common Exception.

Program 10.3

Program to trap and handle *NumberFormatException* (when undesired and invalid number format is encountered).

```
import java.io.*;
public class Program14_3 {
```

```

static boolean valid ;
private static BufferedReader stdin = new BufferedReader
    (new InputStreamReader(System.in)) ;
public static void main(String[ ] args) throws IOException
{
    int n = 0;      See, try { } block surrounds the statement which may
                    cause exception. Here it is z = x / 4 ; as it may cause
                    divide-by-zero exception.
    try { (Hand)
        System.out.print("Enter an integer: ") ;
        //Reading below line of text from the user.
        int input = Integer.parseInt(stdin.readLine());
        valid = true;    //this statement only executes if the string
                        //entered does not cause a NumberFormatException
    }
    catch(NumberFormatException e) {
        //A NumberFormatException occurred, print an error message
        System.out.println(e.getMessage() + " is not a valid format for an integer. ");
    }
}

```

Catch block immediately follows the try block with the exception type it handles

The output produced by above program is as follows :

Note <Exceptionidentifier>. getMessage() returns the input that caused the exception.

Enter an integer: xyz
For input string: "xyz" is not a valid format for an integer.

Although, you can get a complete chapter on exception handling, but here we are limiting ourselves at this only as detailed discussion of Java exception handling is beyond the scope of this book.

10.4 Wrapper Classes

We know that Java's primitive datatypes are data values and not class objects. But sometimes you may encounter situations where numerical values are needed but in the form of objects. Java solves this problem by providing *wrapper classes*.

Wrapper classes are part of Java's standard library `java.lang` and these convert primitive datatypes in an object. To be more specific, a wrapper class wraps a value of primitive type in an object. Java provides following wrapper classes

Datatype	Wrapper Class	Datatype	Wrapper Class
boolean	Boolean	char	Character
byte	Byte	short	Short
int	Integer	long	Long
float	Float	double	Double

Clearly notice the difference. The primitive datatypes are all in lower-case letters and wrapper class names' first letter is always capital letter e.g., **byte** is datatype whereas **Byte** is its wrapper class. You can create an object from wrapper class as follows :

```
Byte b = new Byte(5) ;           // a Byte object created having value 5.
Short s = new Short(170) ;       // a short object created with value 170.
```

Now these **b** and **s** objects (of **Byte** and **Short** types respectively) can be used when desired.

The Wrapper class wraps a value of primitive type in an object. In addition, this class provides several methods for conversion between primary data type and a String, as well as other constants and methods useful when dealing with a byte.

The Wrapper classes also provide various tools such as constants (the smallest and largest possible **int** value, for example) and static methods. You will often use wrapper methods to convert a number type value to a string or a string to a number type (see below).

The wrapper constructors create class objects from the primitive types. For example, for a double floating point number "d" :

```
double d = 5.0 ;
Double aD = new Double(d) ;
```

a **Double** wrapper object is created by passing the **double** value in the **Double** constructor argument.

In turn, each wrapper provides a method to return the primitive value

```
double r = aD.doubleValue( ) ;
```

Each wrapper has a similar method to access the primitive value : **integerValue()** for **Integer**, **booleanValue()** for **Boolean**, etc.

Reasons for wrapper around the primitives may be summarized as :

- ◆ Converting from character strings to numbers and then to other primitive data types.
- ◆ A way to store primitives in an object.

In earlier lines, we have mentioned how to convert strings to primitive types using methods of wrapper classes e.g., **Integer.parseInt()** converts a string into integer ; **Double.parseDouble()** converts a string into double ; **Byte.parseByte()** converts a string to byte, and so on.

```
strTest = "10" ;           int intMy = Integer.parseInt(strTest) ;
strTest = "10.5" ;          double dblMy = Double.parseDouble(strTest) ;
```

Now consider some examples involving wrapper classes that convert primitive values into strings :

```
intMy = 10 ;           String strTest = Integer.toString(intMy) ;
dblMy = 10.0 ;          String strTest = Double.toString(dblMy) ;
```

Appendix B lists various methods in different wrapper classes of Java.

10.5 Working with Strings

In Java, you can work with character data i.e., single character or group of character i.e., strings in three different ways.

Java offers *three* classes to work with character data.

- (i) *Character class* whose instances or objects can hold single character data. This class offers many methods to manipulate or inspect single-character data.
- (ii) *String class* whose instances or objects can hold unchanging string (immutable string) i.e., once initialized its contents cannot be modified.
- (iii) *StringBuffer class* whose instances or objects can hold (mutable) strings that can be changed or modified.

We are not going to discuss *Character class*, which is a wrapper class.

10.5.1 Creating Strings

Strings in Java are created by declaring object of String type class and initializing it with a string literal e.g.,

```
String name = "I am a student" ;
```

You can also create String objects as you would any other Java object i.e., using new operator with a constructor e.g.,

```
String yourname = new String("I do not know your name") ;
```

10.5.2 Creating StringBuffers

StringBuffer objects are always created with new operator. But there are three ways in which you can do so :

- (i) `StringBuffer sBuffer = new StringBuffer() ;`
- (ii) `StringBuffer strBuffer1 = new StringBuffer("First") ;`
- (iii) `int n = 15 ;`
`StringBuffer strBuffer2 = new StringBuffer(n) ;`

The *first* method creates an empty StringBuffer object namely *sBuffer*.

The *second* method creates a StringBuffer object namely *StrBuffer* and initializes it with string value "first" ; the third method, on the other hand, reserves memory to hold 15 characters in StringBuffer object named *strBuffer2*.

The *third* declaration statement creates the StringBuffer with an initial capacity equal to n (=15 here) number of characters. This ensures only one memory allocation for test because it's just big enough to contain the characters that will be copied to it. By initializing the string buffer's capacity to a reasonable first guess, you minimize the number of times memory must be allocated for it. This makes your code more efficient because memory allocation is a relatively expensive operation.

10.5.3 Accessor Methods

Methods used to obtain information about an object are known as *accessor methods*. The class **String** provides many *accessor methods* that may be used to perform operations on strings. The Table (Table 10.2) lists some most used accessor methods of String class.

Table 10.2 Some most used accessor methods of String class

Method prototype	Description
<code>char charAt(int index)</code>	Returns the character at the specified index.
<code>int capacity()</code>	Returns maximum no. of characters that can be entered in the current string object(<code>this</code>) i.e., its capacity.
<code>int compareTo(String1, anotherString)</code>	Compares two strings lexicographically ¹ .
<code>String concat(String str)</code>	Concatenates the specified string to the end of <code>this</code> string (current <i>String</i> object) string.
<code>str1 + str2</code>	Concatenation operator (i.e., +), achieves same as <code>concat</code> method.
<code>boolean endsWith(String str)</code>	Tests if the <code>this</code> string (current <i>String</i> object) ends with the specified suffix (<code>str</code>).
<code>boolean equals(String str)</code>	Compares the <code>this</code> string (current <i>String</i> object) to the specified object <code>str</code> .
<code>boolean equalsIgnoreCase(String str)</code>	Compares the <code>this</code> string (current <i>String</i> object) to <code>str</code> , ignoring case considerations.
<code>int indexOf(char ch)</code>	Returns the index ² within the <code>this</code> string (current <i>String</i> object) of the first occurrence of the specified character.
<code>int lastIndexOf(char ch)</code>	Returns the index within the <code>this</code> string of the last occurrence of the specified character.
<code>int length()</code>	Returns the length of the <code>this</code> string.
<code>String replace(char oldChar, char newChar)</code>	Returns a new string resulting from replacing all occurrences of <code>oldChar</code> in the <code>this</code> string with <code>newChar</code> .
<code>boolean startsWith(String str)</code>	Tests if the <code>this</code> string starts with the specified suffix (<code>str</code>).
<code>String substring(int beginIndex, int endIndex)</code>	Returns a new string that is a substring of the <code>this</code> string.
<code>String toLowerCase()</code>	Converts all of the characters in the <code>this</code> String to lower case.
<code>String toString()</code>	Returns the string itself.
<code>String toUpperCase()</code>	Converts all of the characters in the <code>this</code> String to upper case.
<code>String trim()</code>	Removes white space from both ends of the <code>this</code> String.
<code>String valueOf(all types)</code>	Returns string representation of the passed argument e.g., 12 represented as "12"

1. Lexicography is the way dictionaries are written i.e., alphabetical order to words.

2. The index of first character is 0, second's is 1, third's is 2, and so on.

2. The index of first character is 0, second's is 1, third's is 2, and so on.

There are many more and also variations on the methods presented in Table 10.2. Following example program 10.4 illustrates the usages of many of above string methods.

Program 10.4

Program to illustrate the usage and functionality of many String methods.

```
class StringEx {
// ----- METHODS Follow -----
/* Method using various Java functions */
public void Test( ) {
    String name1 = "\u0052\u0041\u004D\u0041\u004E" ;
    String name2 = "ARORA" ;
    String name3 = "Vincent Roberts" ;
    // Sequence of method calls illustrating string manipulation
    lengthOfString(name1) ; // #1
    charcaterAtIndexN(name1) ; // #2
    compareStrings(name1,name2) ; // #3
    concatenation1(name1,name2) ; // #4
    concatenation2 (name1,name2) ; // #5
    occurrencesOfN(name3) ; // #6
    subString(name3) ; // #7
    replaceCharacter(name3) ; // #8
    caseConversion(name1) ; // #9
}

/* #1 LENGTH OF STRING : Example method to illustrate use of length method */
public static void lengthOfString(String name)
{ System.out.println("LENGTH OF STRING") ;
  System.out.println("Length of string " + name + " is " + name.length( )) ;
}

/* #2 CHARACTER AT INDEX N : Example method illustrating use of
   charAt method. */
public static void charcaterAtIndexN(String name)
{ System.out.println("\nCHARACTER AT INDEX N") ;
  System.out.println("At 0 the character is" + name.charAt(0)) ;
  System.out.println("At 3 the character is" + name.charAt(3)) ;
}

/* #3 COMPARE STRINGS : Example method illustrating use of compareTo
   and equals methods */
public static void compareStrings(String name1, String name2)
{
    System.out.println("\nCOMPARE STRINGS") ;
    // Using the compareTo method
    System.out.println("Compare " + name1 + " and " + name2 + " = " +
        name1.compareTo(name2)) ;
}
```

```

System.out.println("Compare" + name2 + " and " + name1 + " = " +
    name2.compareTo(name1)) ;
System.out.println("Compare" + name2 + " and " + name2 + " = " +
    name2.compareTo(name2)) ;
System.out.print(name1 + " and " + name2 + " are ") ;
// Using the compareTo method within an if-else statement
if (name1.compareTo(name2) == 0)
    System.out.println("the same") ;
else
    System.out.println("not the same") ;
// The equals method
System.out.println("Equals" + name2 + "and" + name1 + "=" +
    name2.equals(name1)) ;
System.out.println("Equals" + name2 + "and" + name2 + "=" +
    name2.equals(name2)) ;
}
/* #4 CONCATENATION 1 : Example method illustrating use of concat method */
public static void concatenation1(String name1, String name2)
{
    System.out.println("\nCONCATENATION 1") ;
    name1 = name1.concat(" ") ;
    name1 = name1.concat(name2) ;
    System.out.println("name1 =" + name1) ;
}
/* #5 CONCATENATION 2 : Example method illustrating use of + (concatenation)
   operator. Same effect as above. */
public static void concatenation2(String name1, String name2)
{
    System.out.println("CONCATENATION 2") ;
    name1 = name1 + " " + name2 ;
    System.out.println("name1 =" + name1) ;
}
/* #6 OCCURANCES OF N : Example method illustrating use of indexOf and
   lastIndexOf method */
public static void occurrencesOfN(String name)
{
    System.out.println("OCCURANCES OF n") ;
    // Find first and last occurrences of 'n' in argument
    System.out.println("First occurrence of 'n' in" + name +
        "is at index" + name.indexOf('n')) ;
    System.out.println("Last occurrence of 'n' in" + name +
        "is at index" + name.lastIndexOf('n')) ;
    // What happens if we are looking for the index of something
    // that does not exist in the given string ?
    System.out.println("First occurrence of 'Z' in" + name +
        "is at index" + name.indexOf('Z')) ;
}

```

```

/* #7 SUBSTRINGS : Example method illustrating use of substring method */
public static void subString(String name1)
{
    System.out.println("\nSUBSTRINGS") ;
    // Substring from index 4 onwards
    System.out.println("Substring from index 4 onwards =" + name1.substring(4)) ;
    // Substring comprising all of input
    String name2 = name1.substring(0,name1.length()) ;
    System.out.println("Entire copy of" + name1 + "=" + name2) ;
    // Substring from N to M
    name2 = name1.substring(0,2) ;
    System.out.println("Substring from index 0 to 2 =" + name2) ;
    // Substring made up of name2 from above (first three characters
    // of name1) plus substring from the first occurrence of a space
    // (' ') in name 1 to three characters beyond it.
    name2 = name2.concat(name1.substring(name1.indexOf(' '),
    name1.indexOf(' ') + 3)) ;
    System.out.println("name2 =" + name2) ;
}

/* #8 REPLACE CHARACTER : Example method illustrating use of replace method */
public static void replaceCharacter(String name)
{
    System.out.println("\nREPLACE CHARACTER") ;
    // Replace R and O with.
    name = name.replace('R','.') ;
    name = name.replace('O','.') ;
    System.out.println("name =" + name) ;
}

/* #9 CASE CONVERSION : Example method illustrating use of toLowerCase
   and toUpperCase methods */
public static void caseConversion(String name1)
{
    System.out.println("CASE CONVERSION") ;
    // Upper to lower case conversion
    name1 = name1.toLowerCase( ) ;
    System.out.println("name1 =" + name1) ;
    // Lower to upper case conversion
    name1 = name1.toUpperCase( ) ;
    System.out.println("name1 =" + name1) ;
    // Substrings and case conversion
    String name2 = name1.substring(0,1) ;
    String name3 = name1.substring(1,5) ;
    name2 = name2.concat(name3.toLowerCase( )) ;
    System.out.println("name2 =" + name2) ;
}

```

The output produced by above program is as follows :

LENGTH OF STRING

Length of string RAENA is 5

CHARACTER AT INDEX N

At 0 the character is F

At 3 the character is N

COMPARE STRINGS

Compare RAENA and ARORA=5

Compare ARORA and RAENA=-5

Compare ARORA and ARORA=0

RAENA and ARORA are not the same

Equals ARORA and RAENA=false

Equals ARORA and ARORA=true

CONCATENATION 1

name1 =RAENA ARORA

CONCATENATION 2

name1 =RAENA ARORA

OCCURRENCES OF n

First occurrence of 'n' in Vincent Roberts is at index2

Last occurrence of 'n' in Vincent Roberts is at index5

First occurrence of 'z' in Vincent Roberts is at index-1

SUBSTRINGS

Substring from index 4 onwards =ent Roberts

Entire copy of Vincent Roberts=Vincent Roberts

Substring from index 0 to 2 =Vi

name2 =Vi Ro

REPLACE CHARACTER

name =Vincent .oberts

CASE CONVERSION

name1 = raena

name1 = RAENA

name2 = Raena

Let us briefly discuss what happened in above program :

1. The length method returns the length of the string i.e., the number of characters in the string.
2. We used the charAt method to return the character value at a particular position.
3. When using the compareTo method, the method returns an integer indicating the lexicographic (alphabetic) location of the first letter of the first string with the first letter of the second string. If the first letters are both different, integer is returned indicating the relative alphabetic location of the second start letter compared with the first, negative if the second is lexicographically after the first and positive otherwise.

If both strings are identical then a 0 value is returned. If both have the same start letter but are not the same either 32 or -32 is returned according to whether the second is lexicographically before or after the first. The `compareTo` method can thus be usefully incorporated into selection statements (if-else, switch).

4. Alternatively if all we are interested in its direct comparison, we can use the `equals` method which returns a `boolean` value.
5. The concatenation method appends its argument (which must be a string) to the indicated string and returns a *new instance of the class String*. In the example this new string is also called `name1` thus overwriting the previous instance of that name.
6. The concatenation operator (`+`) can be used to produce the same result as the `concat` method.
7. The `indexOf` and `lastIndexOf` methods find occurrences of a particular character starting from the start or end of the string respectively, and return the index to that character. If the occurrence cannot be found, the methods return -1. A variation on the `indexOf` method includes a second argument to indicate a start index for the search.
8. The `subString` method is used to create new instances of the class `String` from existing instances. The new string is specified by giving the required index range within the existing string (inclusive of the start index and exclusive of the end index). If no upper bound is specified for the substring, Java assumes that this is the upper bound of the given string. The method can also be used to create a copy of a string.
9. The `replace` method is used to replace each occurrence of the first argument in a string with the second argument.
10. The `toUpperCase` and `toLowerCase` methods convert a given string to lower case or uppercase characters respectively.

`StringBuffer` class

The `StringBuffer` class, which is part of `java.lang` package, is alternative to the `String` class. A `StringBuffer` object contains a memory block called a *buffer* which may not contain a string.

You have already learnt to create `StringBuffer` objects. Let us now learn about some more facts about `StringBuffer` class.

- ❖ The length of the `String` may not be the same as the length of the buffer. (Fig. 10.4)
- ❖ The length of the buffer is referred to as the capacity of the `StringBuffer` object.
- ❖ You can change the length of a `String` in a `StringBuffer` object with the `setLength()` method.
- ❖ When the `StringBuffer` object's length is longer than the `String` it holds, the extra characters contain '\u0000'.
- ❖ If you use the `setLength()` method to specify a length shorter than its `String`, the string is truncated.

Figure 10.4 illustrates the difference between `length()` and `capacity()` methods.

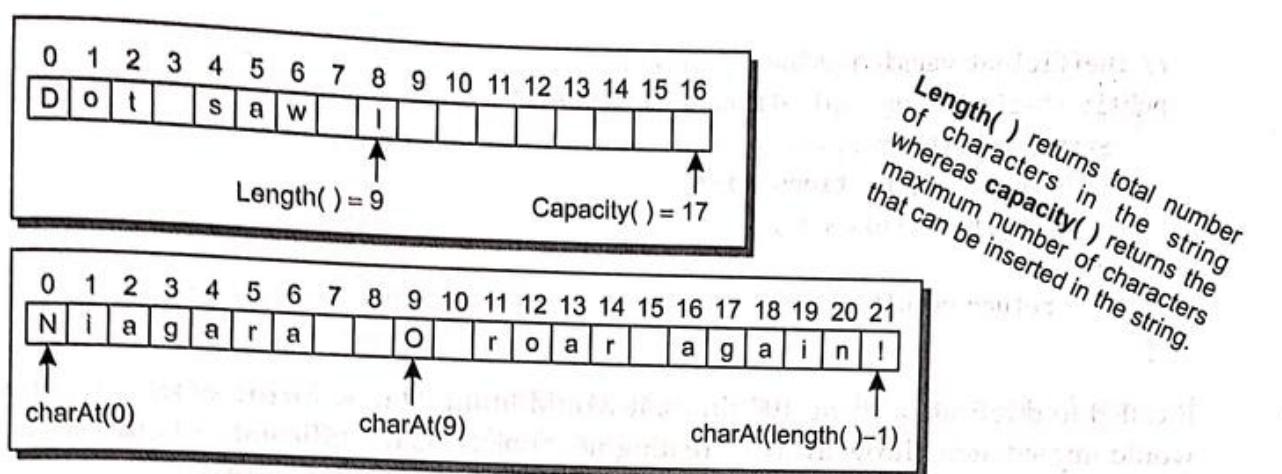


Figure 10.4 Functionality of some string function.

Using **StringBuffer** objects provide more flexibility than **String** objects because you can *insert* or *append* new contents into a **StringBuffer**. All the above given string functions can also be used with **StringBuffer** class. However, there are some additional methods that you can use with **StringBuffer** objects. Let us talk about these additional **StringBuffer** methods.

- ◆ **append(x) method.** Lets you add *x* characters to the end of a **StringBuffer** object.
- ◆ **insert(offset, x) method.** Lets you add *x* characters at a specified location *offset* within a **StringBuffer** object.
- ◆ **setCharAt(index, c) method.** Alters just one character. It replaces character at *index* with *c* in a **StringBuffer**.
- ◆ **delete (beg, end) method.** Deletes characters at index *beg* thru *end*.
- ◆ **setLength(n) method.** Sets the length of the content to *n* by either truncating current content or extending it with the null character ('\u0000'). Use *sb.setLength(0)* ; to clear a string buffer.
- ◆ **reverse() method.** Reverses the contents of **StringBuffer**

An interesting aspect of the **append()** and **insert()** methods is that the parameter may be of any type. These methods are overloaded and will perform the default conversion for all primitive types and will call the **toString()** method for all objects.

Chaining calls

Some **StringBuffer** methods return a **StringBuffer** value (e.g., **append()**, **insert()**, ...). In fact, they return the same **StringBuffer** that was used in the call. This allows *chaining* of calls e.g.,

```
sb.append("x =").append(x).append(", y =").append(y);
```

If *x* is 5 and *y* is 2, the above code will result into *x = 5, y = 2*.

Efficiency of **StringBuffer** compared to **String**

Because a **StringBuffer** object is *mutable* (it can be changed), there is no need to allocate a new object when modifications are desired. Recall that if a **String** is modified, new **String** object is created to hold changes. The original string remains unchanged. (We described it in chapter 8 – functions, program 8.5). For example, consider a method which duplicates strings the requested number of times.

```
// Inefficient version using String.
public static String dupl(String s, int times) {
    String result = s;
    for(int i = 1; i < times; i++) {
        result = result + s;
    }
    return result;
}
```

If called to duplicate a string 100 times, it would build 99 new **String** objects, 98 of which it would immediately throw away ! Creating new objects is not efficient. A better solution is to use **StringBuffer**.

```
// More efficient version using StringBuffer.
public static String dupl(String s, int times) {
    StringBuffer result = new StringBuffer(s);
    for(int i = 1; i < times; i++) {
        result.append(s);
    }
    return result.toString();
}
```

This creates only two new objects, the **StringBuffer** and the *final String* that is returned. **StringBuffer** will automatically expand as needed. These expansions are costly however, so it would be better to create the **StringBuffer** the correct size from the start as shown in following code fragment

```
StringBuffer result = new StringBuffer(s.length() * times);
```

Now consider the following program which tests whether a string is palindrome or not.

Program 10.5

Program to check whether a string is palindrome or not.

```
import java.io.*;
public class Palindrome {
    /*
     * Program execution starts here. The String must be surrounded in
     * quotes, if it has any blank spaces. Otherwise, each word will be
     * tested to see if it is a palindrome. */
    public static void main(String args[ ]) throws IOException {
        // run sample tests
        showPalindrome("mom");
        showPalindrome("dad");
        showPalindrome("radar");
        showPalindrome("Mom");
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader stdin = new BufferedReader(isr);
```

```

// Now ask the user to type a string and then test it
try {
    System.out.println("\nEnter a string for palindrome test");
    String input = stdin.readLine();
    showPalindrome(input);
}
catch(Exception e) { }
} // end main method

/* Displays results of testing the String with each type of palindrome. */
public static void showPalindrome(String s) {
    System.out.print(s);
    if(isPalindrome(s))
        System.out.println(": IS a palindrome!");
    else if(isPalindrome2(s))
        System.out.println(": IS a palindrome if you ignore case!");
    else
        System.out.println(": IS NOT a palindrome!");
} // end showPalindrome

/* Returns true if String of characters is a palindrome. */
public static boolean isPalindrome(String s) {
    StringBuffer reversed = (new StringBuffer(s)).reverse();
    return s.equals(reversed.toString());
} // end isPalindrome

/** Returns true If String is a palindrome when case is ignored. */
public static boolean isPalindrome2(String s) {
    StringBuffer reversed = (new StringBuffer(s)).reverse();
    return s.equalsIgnoreCase(reversed.toString());
} // end isPalindrome2
} // end Palindrome

```

The output produced by above code is as follows :

```

mom: IS a palindrome!
dad: IS a palindrome!
radar: IS a palindrome!
Mom: IS a palindrome if you ignore case!
Enter a string for palindrome test
rakesh
rakesh: IS NOT a palindrome!

```

10.6 Packages in Java

When a largish program is to be written, it is often advantageous to divide the program up into chunks (modules) and compile the parts separately. This option is particularly desirable when several programmers are jointly developing a large program. Such chunks of code are

called *packages* in Java. Each package is stored in a directory (folder) which has the same name as the package. Thus a package called LocalPackages will be stored in a directory called LocalPackages.

We have already seen how we can import Java's standard API packages, such as the java.io package, into a program and then use the classes defined in this package. We can also write our own *user-defined* packages containing implementations of our own classes.

Note To use the classes and interfaces defined in one package from within another package, you need to import the package using import command.

10.6.1 Importing Packages and their Classes

To import a specific class or interface into the current file (like the Circle class from the graphics package created in the previous section) use the import statement :

```
import graphics.Circle ;
```

The import statement must be at the beginning of a file before any class or interface definitions and makes the class or interface available for use by the classes and interfaces defined in that file.

If you want to import all the classes and interfaces from a package, for instance, the entire graphics package, use the import statement with the asterisk '*' wildcard character e.g.,

```
import graphics.* ;
```

If you try to use a class or interface from a package that has not been imported, the compiler will issue a fatal error :

```
testing.java : 4 : Class Date not found in type declaration.  
Date date ;  
^
```

Note that only the classes and interfaces that are declared to be public can be used by classes outside of the package that they are defined in.

10.6.2 Using Dates and Times (Date and Calendar Objects)

This section briefly describes how to use and manipulate dates and times in Java. In order to use dates and times, *java.util* package is to be imported in your program. In this section, we are going to talk about following two classes that can be used for manipulating, using date and times in your programs :

Class	Purpose
(i) <code>java.util.Date</code>	a specific instant in time.
(ii) <code>java.util.Calendar</code>	conversion of Date to integer fields.

Let us know some more about these classes.

Note The default package (a package with no name) is always imported for you. The runtime system automatically imports the *java.lang* package for you as well i.e., you need not write an import statement for *java.lang* package.

Date (i.e., java.util.Date) class

A **Date object** represents a specific instant in time. In old versions of Java the **Date object** was used for lots of things, but now **Calendar** does most of the work. Most of the methods of Date class are deprecated !

The non-deprecated use of Date class includes :

- ❖ Conversion to/from the number of milliseconds since *Jan 1, 1970*.
- ❖ Comparison of dates (after, before, equals, ...)

Following example program (10.6) illustrates the use of Date class for printing current Date/Time.

Program 10.6

Program to illustrate the use of Date class for printing current Date/Time.

```
import java.util.* ;
class DatePlay {
    static public void main(String[ ] args) {
        Date d = new Date() ;
            //create Date object and by default initialize it with system date
        System.out.println("The date is " + d) ;
    }
}
```

The output produced is

The date is Thu Feb 09 17:44:29 GMT+05:30 2006

In the above program the statement

`Date d = new Date() ;`

creates **Date object d** and initializes it with system date automatically.

Calendar class (java.util.Calendar class)

This class is an abstract class whose object is created with the help of following (*Object Factory*) method :

```
static Calendar getInstance( ) ;
e.g.,   Calendar calCurrent = Calendar.getInstance( ) ;
                    // create a Calender object namely calCurrent
```

The **Calendar object** can set/get integer values for day, month, year, hour, minute, second, day of the week, etc., using one of the following two (overloaded) methods :

```
set(int field, int value) ;           // set a specified field with specific value
get(int field) ;                   // obtain value of specific field
```

You have to know what the value of field should be to get what you want. The **Calendar class** provides some static int fields that make this possible.

These **Calendar Fields** are as follows :

Calendar.DATE	Stores numeric day
Calendar.MONTH	Stores numeric month
Calendar.YEAR	Stores numeric year
Calendar.DAY_OF_MONTH	Stores day of the month
Calendar.DAY_OF_WEEK	Stores character week
Calendar.DAY_OF_YEAR	Stores day of the year
Calendar.HOUR	Stores hh part of time hh:min:ss.ms
Calendar.MINUTE	Stores min part of time hh:min:ss.ms
Calendar.SECOND	Stores ss part of time hh:min:ss.ms
Calendar.MILLISECOND	Stores ms part of time hh:min:ss.ms

There are lots more, but here we are not going to discuss them.

Let us now learn to use *Calendar* class with the help of following example program 10.7.

Program 10.7

Program to illustrate the use of *Calendar* class for printing current Date.

```
import java.util.* ;
class CalPlay {
    static public void main(String[ ] args) {
        Calendar c = Calendar.getInstance( ) ;
        System.out.println("Today is " +
            (c.get(Calendar.MONTH) + 1) + "/" +
            c.get(Calendar.DATE) + "/" +
            c.get(Calendar.YEAR)) ;
    }
}
```

The output produced is

Today is 01/15/2016

See the date is printed in *mm/dd/yyyy* format.

10.6.3 Packages in Java

Java contains an extensive library of pre-written classes you can use in your programs. These classes are divided into groups called *packages*.

Various packages included in Java 1.1 are given below :

- **java.applet** Java applet package
- **java.awt** Java Abstract Window Toolkit package
- **java.io** Java Input/Output package
- **java.lang** Java Language package
- **java.net** Java Networking package
- **java.util** Java Utility package

Note All input and output related classes are part of **java.io** package.

Each package defines a number of classes, interfaces, exceptions, and errors.

10.6.4 User Defined Packages

Packages are groups of related classes and interfaces and provide a convenient mechanism for managing a large set of classes and interfaces and avoiding naming conflicts. In addition to it using Java's *package* statement.

Suppose that you are implementing a group of classes that represent a collection of graphic objects such as *circles*, *rectangles*, *lines*, and *points*. If you want to make these classes available to other programmers, you bundle them together into a package called, say, **graphics** and give the package to the programmers (along with some reference documentation as to what the classes and interfaces do and what their public programming interfaces are).

In this way, other programmers can easily determine what *your group of classes are for, how to use them, and how they relate to one another and to other classes and packages*. Also, the names of your classes won't conflict with class names in other packages because the classes and interfaces within a package are referenced in terms of their package (technically a package creates a new namespace.)

You create a package using the package statement as shown in following example :

```
package graphics ;
class Circle {
    ...
}
class Rectangle {
    ...
}
```



This statement creates a package called **graphics**

The first line in the preceding code sample creates a package called **graphics**. All the classes and interfaces defined in the file containing this statement in it are members of the **graphics package**. So, *Circle*, and *Rectangle* are all members of the new **graphics** package.

Thus, to create a package and then classes inside this package, you need to do this :

- ❖ declare the package name in the first statement in your program.
- ❖ define the class(es) inside this package below the package declaration statement.

Please note that a class can have only one package declaration.

Let us create a user defined package and then use it practically. In the following lines, we are creating a user-defined package namely **personalCalculator** that contains two classes in it.

```
package personalCalculator ; // this package contains two classes
public class BasicCalculations {
    public int add(int a, int b) {
        return a + b;
    }
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

```

        public int multiply(int a, int b) {
            return a * b;
        }

        public int divide(int a, int b) {
            return a / b;
        }

        public static void main(String args[]) {
            // using the class methods inside own class, just like other classes
            BasicCalculations obj = new BasicCalculations();
            System.out.println("Add method " + obj.add(25, 7));
            System.out.println("Multiply method " + obj.multiply(25, 7));
        }
    }

    class BasicCalculations2 {
        public int modulus(int a, int b) {
            int m, q;
            q = a / b;
            m = a - (q * b);
            return m;
        }

        public int PositivePower(int a, int b) {
            // this method calculates only the positive powers of a number
            // and returns -99999 for negative value of power
            int res = 1;
            if (b == 0) return res;
            else if (b > 0) {
                for (int i = 0; i < b; i++) {
                    res = res * a;
                }
                return res;
            }
            else
                return -99999; // -99999 indicates illegal
        }
    }
}

```

Here, the methods
of class
BasicCalculations
are being used
within own class

Now let's see how to use this package in another program.

```

import personalCalculator.*;
public class Check
{
    public static void main(String args[])
    {
        // using the class methods inside own class
    }
}

```

All calluses of package
personalCalculator
imported

```

BasicCalculations obj = new BasicCalculations();
Objects created from
Imported classes

System.out.println("Add method " + obj.add(25, 7));
Methods of class
BasicCalculations
used here

System.out.println("Multiply method " + obj.multiply(25, 7));

BasicCalculations2 obj2 = new BasicCalculations2();
Method of class
BasicCalculations2
used here

System.out.println("Modulus method " + obj2.modulus(25, 7));
}

}

```

You can use the methods of a class (which is part of a package) within itself like you do for any other class. (Refer to code of class *BasicCalculations* – member of package *personalCalculator*)

To use the classes **BasicCalculations** and **BasicCalculations2**, we have imported the package **personalCalculator** using the statement :

```
import personalCalculator.*;
```

The above statement will import all the classes of package **personalCalculator** and now you can call any of the methods inside these classes as you can see in code of class **Check** above.

If, however, you want to import only one class from a package, you can import only that class by specifying the class name with its package name as shown below :

```
import personalCalculator.BasicCalculations;
```

The *.class* files generated by the compiler when you compile the file that contains the source for *Circle*, and *Rectangle* must be placed in a directory named *graphics* somewhere in your CLASSPATH. Your CLASSPATH is a list of directories that indicate where on the file system you've installed various compiled Java classes and interfaces. When looking for a class, the Java interpreter searches your CLASSPATH for a directory whose name matches the package name of which the class is a member. The *.class* files for all classes and interfaces defined in the package must be in that package directory.

Your package names can have multiple components (separated by periods). In fact, the Java package names have multiple components : *java.util*, *java.lang*, and so on. Each component of the package name represents a directory on the file system. So, the *.class* files for *java.util* are in a directory named *util* in a directory named *java* somewhere in your CLASSPATH.

Let Us Revise

- ❖ Java includes predefined classes in the form of packages, also called Java class libraries.
- ❖ *System.in* and *System.out* refer to input stream and output stream respectively, managed by *System* class.
- ❖ Streams can either be byte oriented that reads bytes of data or binary data. Or the streams can be character oriented that use a character encoding scheme to read character.

- ❖ Character Oriented IO is performed through Readers and Writers.
- ❖ Byte Oriented IO is performed through data streams.
- ❖ Way of handling anomalous situations in a program run is known as exception handling.
- ❖ The throws keyword automatically informs the error occurrence so that it can be trapped.
- ❖ The try block is for enclosing the code wherein exceptions can take place.
- ❖ The catch block traps the exception and handles it.
- ❖ The throw keyword forces an exception.
- ❖ Wrapper classes wrap the value of a primitive type in an object.
- ❖ Java has these wrapper classes – Boolean, Byte, Integer, Float, Character, Short, Long and Double.
- ❖ The String objects are immutable (unchangeable) but StringBuffer objects are mutable.
- ❖ Related classes and interfaces are grouped together in the form of package.
- ❖ Packages and their classes are imported through import command.
- ❖ The package statement in a program creates a package and makes all following classes, its part.

Solved Problems

1. What is the difference between byte oriented IO and character oriented IO ? How are these two performed in Java ?

Solution. Byte oriented IO reads bytes of data or binary where there is no notion of data types. Character oriented IO, on the other hand, performs IO which is specially character oriented.

In Java, Byte Oriented IO is performed through *data streams* whereas character oriented IO is performed through *Readers and Writers*.

2. What is an Exception ?

Solution. Exception in general refers to some contradictory or unusual situation which can be encountered while executing a program.

3. When is Exception Handling required ?

Solution. The exception handling is ideal for :

- ☒ processing exceptional situations
- ☒ processing exceptions for components that cannot handle them directly
- ☒ processing exceptions for widely used components that should not process their own exceptions
- ☒ large projects that require uniform error-processing.

4. What is the function of catch block in exception handling ? Where does it appear in a program ?

Solution. A catch block is a group of Java statements that are used to handle a raised exception. Catch blocks should be placed after each try block.

5. What are the advantages of exception handling ?

Solution. Advantages of exception handling are :

- (i) Exception handling separates error-handling code from normal code.
- (ii) It clarifies the code and enhances readability.
- (iii) It stimulates consequences as the error-handling takes place at one place and in one manner.
- (iv) It makes for clear, robust, fault-tolerant programs.

6. When do we need multiple catch handlers ?

Solution. Sometimes program has more than one condition to throw exceptions. In such cases, one can associate more than one catch statement with a try.

7. Differentiate between *String* and *StringBuffer* objects.

Solution. The string objects of Java are immutable i.e., once created, they cannot be changed. If any change occurs in a string object, then original string remains unchanged and a new string is created with the changed string.

StringBuffer objects are mutable, on the other hand. That is, these objects can be manipulated and modified as desired.

8. Given a package named *EDU.Student*, how would you import a class named *Test* contained in this package ? Write one line statement.

Solution. `import EDU.Student.Test ;`

9. There are three classes that implement the *DataInput* and *DataOutput* interfaces. Two of them are *DataInputStream* and *DataOutputStream*. Which is the third one ?

Solution. `RandomAccessFile` class

10. What will be the output of the following code snippet when combined with suitable declarations and run ?

```
StringBuffer city = new StringBuffer ("Madras") ;
StringBuffer string = new StringBuffer ( ) ;
string.append(new String(city)) ;
string.insert(0, "Central") ;
String.out.println(string) ;
```

Solution. CentralMadras.

11. Give the output of the following program :

```
class MainString
{
    public static void main(String args[ ])
    {
        StringBuffer s = new StringBuffer("String") ;
        if(s.length( ) > 5) &&
            (s.append("Buffer").equals("X"))
        ; // empty statement
        System.out.println(s) ;
    }
}
```

Solution. `StringBuffer`.

12. Consider the following try.... catch block :

```
class TryCatch
{
    public static void main (String args [ ] )
    {
        try
        {
            double x = 0.0 ;
            throw(new Exception("Thrown")) ;
        }
    }
}
```

```

        catch(Exception e)
        {
            System.out.print("Exception caught");
            return;
        }
        finally {
            System.out.println("finally");
        }
    }
}

```

What will be the output?

Solution. Exception caught finally.

13. What is the output of the following code fragment if "abc" is passed as argument to the function?

```

public static void func(String s1) {
    String s = s1 + "xyz";
    System.out.println("s1 =" + s1);
    System.out.println("s =" + s);
}

```

Solution. s1 = abc
 s = abcxyz

Glossary

Exception An anomalous situation encountered by the program.

Syntax error Programming language's grammar rules violation error.

Compile time error Error that the compiler can find during compilation.

Run time error Error during program execution.

Accessor Methods Methods used to obtain information about an object.

Java package Group of related classes and interfaces.

A ssignments

TYPE A : VERY SHORT/SHORT ANSWER QUESTIONS

1. Predefined classes are available in the form of _____.
2. Name the package you need to import for performing input and output.
3. Which streams are by default available in a Java program?
4. Which class stores the information about the connection between an input device and the computer or program?

5. Name the class that translates data bytes received from Input Stream objects into a stream of characters.
6. Which class buffers the input received from an InputStreamReader object ?
7. Which package is by default imported in every Java program ?
8. Name the block that encloses the code that may encounter anomalous situations.
9. In which block can you throw the exception ?
10. Which block traps and handles an exception ?
11. Can one catch block sufficiently trap and handle multiple exceptions ?
12. Which block is always executed no matter which exception is thrown ?
13. NumberFormatException is subclass of which exception class ?
14. IOException is subclass of which exception class ?
15. What is the role of (i) throws keyword ? (ii) throw keyword ?
16. Why is exception handling necessary ?
17. What is the difference between equals() and equalsIgnoreCase() string functions ?
18. What is the difference between length() and capacity() string functions ?
19. The default package is a package without any name and is imported for you. (T/F)
20. Which command creates a package in Java ?

TYPE B : LONG ANSWER QUESTIONS

1. What are two ways of obtaining input in Java ?
2. How are Readers and Writers different from other streams ?
3. Write a short note on exception Handling in Java.
4. How do we define a **try** block ?
5. How do we define a **catch** block ?
6. List some of the most common types of exceptions that might occur in Java. Give examples.
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.
8. What is a **finally** block ? When and how is it used ? Give a suitable example.
9. Write a program to do the following :
 - (a) To output the question "Who is the inventor of Java" ?
 - (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.
10. 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.
11. Write a program, which will read a text and count all occurrences of a particular word.
12. What is a package ? How do we tell Java that we want to use a particular package in a file ?
13. How do we design a package ?
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).

15. Write a program to convert the given temperature in Fahrenheit to Celsius using the following conversion formula :

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

16. 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 (m/s), a is the acceleration (m/s^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 .

17. What are the applications of wrapper classes ?

Arrays

In This Chapter

11.1 Introduction

Till now you have been working with data in Java program in simple fashion. But now, I want to ask a question from you – how would you create classes / objects for a situation wherein you have to deal with a collection of similar data items ? For example, you have to handle marks of 20 students. See, you have to handle 20 *marks* variables. How would you do so ?

Well, you need not worry about it. To simplify things for you, Java offers a reference data type called *arrays*. An array can hold several values of the same type.

Arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. Arrays are a way to group a number of items into a larger unit. Arrays can have data items of simple types like *int* or *float*, or even of user-defined types like structures and objects.

An **Array** is a collection of variables of the same type that are referenced by a common name.

- 11.1 Introduction
- 11.2 Need for Arrays
- 11.3 Types of Arrays
- 11.4 Searching in 1-D Arrays
- 11.5 Sorting
- 11.6 Arrays vs. Objects
- 11.7 Advantages and Disadvantages of Arrays
- 11.8 Solution of Simultaneous Linear Equations

11.2 Need for Arrays

Why do we need arrays? We will illustrate this with the help of an example. Let us consider a situation, where we have to write a program which can accept marks of 50 students of a class, calculate their average marks and then calculate the difference of each student's marks with average marks.

If we solve this problem by making use of variables, we need 50 variables to store student's marks, 1 variable to store average marks and 50 more variables to store difference marks i.e., in all we need 101 variables each having a different name. Now, remembering and managing these 101 variables is not an easy task and it will make the program a complex and ununderstandable program.

But if we declare two arrays each having 50 elements; one for students' marks and another for difference marks, and 1 variable for storing average marks. Now, we only have to remember three names i.e., two arrays' names and a variable name (for average marks). Elements of these arrays will be referred to as *arrayname[n]*, where *n* is the element number in the array (refer to Fig. 11.1). Writing such a program would not only be simple but also very easy to code and understand. Therefore, arrays are very much useful in a case where *quite many elements of the same (data) types need to be stored and processed*.

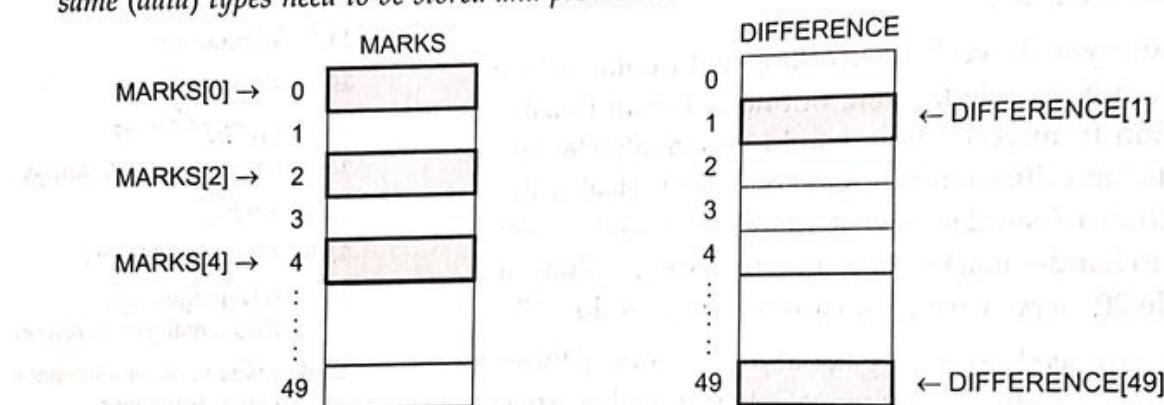


Figure 11.1 One-dimensional arrays.

The element numbers in [] are called *subscripts* or *indices*. The *subscript*, or *index* of an element designates its position in the array's ordering.

11.3 Types of Arrays

Arrays are of different types :

- (i) *one-dimensional arrays*, comprised of finite homogeneous elements.
- (ii) *multi-dimensional arrays*, comprised of elements, each of which is itself an array.

A two-dimensional array is the simplest of multidimensional arrays. However, Java allows arrays of more than two dimensions. The exact limit (of dimensions), if any, is determined by the compiler you use.

11.3.1 Single Dimensional Arrays

The simplest form of an array is a single dimensional array. The array is given a name and its elements are referred to by their *subscripts* or *indices*. Java array's index numbering starts with 0 (zero).

Like other variables in Java, an array must be defined before it can be used to store information. Like other definitions, an array definition specifies a *variable type* and a *name* along with one more feature *size* to specify how many data items the array will contain. The general form of an array declaration is as shown below :

type array-name[] = new type [size] ;
or type[] arrayname = new type [size] ;

where *type* declares the base type of the array, which is the type of each element in the array. The *array-name* specifies the name with which the array will be referenced and *size* defines how many elements the array will hold. The *size* must be an integer value or integer constant without any sign.

The data type of array elements is known as the **base type** of the array.

Declaring Arrays

Following statements declare an array *marks* of base type **int** and which can hold 50 elements :

int marks [] = new int [50] ;
or int [] marks = new int [50] ;

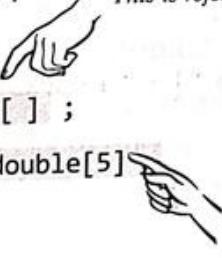
The above statements declare array *marks* which have 50 elements, *marks* [0] to *marks* [49].

An array must be *declared* or created before it can be used. This is a two-step process :

- ❖ First, declare a reference to an array.
- ❖ Then, create the array with the new operator.

For instance :

This is reference to a double type array



```
double x[ ] ;                                            // Create reference
```



```
x = new double[5]                                    // Create array object by assigning memory to it
```

Here an array of 5 doubles is created and its reference is stored in x ;

These steps can be combined on a single line, as shown below :

```
double x[ ] = new double[5] ; // All together i.e., create reference and assign memory
```

Using Arrays

An **array element** may be used in any place where an ordinary variable of the same type may be used. An array element is addressed using the array name followed by a integer subscript in brackets e.g., *a*[2]

The individual element can now be accessed as :

marks [0] will give us the first student's marks ;
marks [1] will give us the second student's marks ;
marks [49] will give us the fiftieth student's marks ;

The following program reads marks of 50 students and stores them in an array *marks*. Arrays are typically used to perform the *same calculation on many different values*. For example, consider the following code that calculates and stores back the square root of array elements:

```
for ( int i = 0 ; i < 100 ; i++ )
{
    a[ i ] = Math.sqrt( a[ i ] ) ;
} // sqrt of 100 values
refers to element of array a at index i e.g., if i = 0,
then it'll refer to a[0] element.
```

Program 11.1

Program to read marks of 50 students and store them under an array.

```
import java.io.DataInputStream ;
class ar1 {
    public static void main(String args[ ]) {
        DataInputStream in = new DataInputStream(System.in) ;
        final int size = 5 ;
        int i ;
        int marks[ ] = new int[size] ;
        try {
            for(i = 0; i < size; i++) {
                System.out.print("Enter marks of student " + (i + 1) + " : " );
                marks[i] = Integer.parseInt(in.readLine()) ;
            }
        } catch (Exception e) { }
        for(i = 0; i < size; i++)
            System.out.println("Marks[" + ( i ) + "] = " + marks[i]) ;
    }
}
```

Sample program run for 5 students is shown below :

```
Enter marks of student1 : 67
Enter marks of student2 : 78
Enter marks of student3 : 90
Enter marks of student4 : 79
Enter marks of student5 : 70
Marks[0] = 67
Marks[1] = 78
Marks[2] = 90
Marks[3] = 79
Marks[4] = 70
```

Note In Java arrays, the element at position 1 has index/subscript 0 ; element at position 2 has index 1 ; element at position 3 has index 2 ; and so on. Thus, position of an element is index +1.

Consider another example program that also uses single-dimensional array.

Program 11.2

Program to accept sales of each day of the month and print the total sales and average sales of the month.

```
import java.io.DataInputStream ;
class ar2 {
    public static void main(String args[ ]) {
        DataInputStream in = new DataInputStream(System.in) ;
        final int size = 30 ;
        int i ;
        float sales[ ] = new float[size] ;
        float avg = 0, total = 0 ;
        try {
            for(i = 0; i < size; i++) {
                System.out.print("Enter sales made on day " + (i + 1) + " : ") ;
                sales[i] = Float.valueOf(in.readLine()).floatValue() ;
                total = total + sales[i] ;
            }
        } catch (Exception e) { }
        avg = total / size ;
        System.out.println("Total sales = " + total) ;
        System.out.println("Average sales = " + avg) ;
    }
}
```

Sample program run for 3 days is shown below :

```
Enter sales made on day 1 : 2000
Enter sales made on day 2 : 1500
Enter sales made on day 3 : 1800
Total sales = 5300.0
Average sales = 1766.6666
```

The above program (11.2) first declares a constant integer *size* with value 30. This constant integer *size* is used to declare the size of the array *sales*. Sales values are read into array elements and simultaneously *total* (*sales*) is calculated. Finally, after calculating average, total sales and average is printed.

Let us take one more example. Given monthly salaries of 100 employees of an organization, you are supposed to find out the number of employees earning more than ₹ 1 lakh per annum. Again we shall be using single dimensional array to hold the employees' salaries.

Program 11.3

Program to count the number of employees earning more than Rs. 10 lakh per annum. The monthly salaries of 100 employees are given.

```
import java.io.DataInputStream ;
class ar3 {
    public static void main(String args[ ]) {
        DataInputStream in = new DataInputStream(System.in) ;
        final int size = 100 ;           // for sample run, we'll make it 5
        int i, count = 0 ;
        float Sal[ ] = new float[size] ;
        float an_sal ;
        try {
```

```

        for(i = 0; i < size; i++) {
            System.out.print("Monthly salary for employee " + (i + 1) + " : ");
            Sal[i] = Float.valueOf(in.readLine()).floatValue();
        }
    } catch (Exception e) { }
    for(i = 0; i < size; i++) {
        an_sal = Sal[i] * 12;
        if(an_sal > 100000)
            count++;
    }
    System.out.println(count + " employees out of " + size +
        " employees are earning more than Rs. 10 Lakh per annum");
}
}

```

A sample program run for 5 employees is shown below :

```

Monthly salary for employee 1 : 12000
Monthly salary for employee 2 : 200000
Monthly salary for employee 3 : 115000
Monthly salary for employee 4 : 15000
Monthly salary for employee 5 : 300000
3 employees out of 5 employees are earning more than Rs. 10 Lakh per annum

```

Here, I would like to mention a related term, **vector**. A **vector** is a mathematical term which refers to the collection of numbers which are analogous *i.e.*, a linear array (one dimensional arrays). But vectors are different from arrays as they can grow in size whereas arrays cannot.

11.3.1A Memory Representation of Single Dimension Arrays

Single-dimension arrays are essentially lists of information of the same type and their elements are stored in contiguous memory location in their index order. For instance, an array *grade* of type *char* with 8 elements declared as

```

char grade [ ] = new char [8];
or as char[ ] grade = new char [8];

```

will have the element *grade* [0] at the first allocated memory location, *grade* [1] at the next contiguous memory location, *grade* [2] at the next, and so forth. Since *grade* is a *char* type array, each element size is 2 bytes (A character size is 2 bytes in Java) and it will be represented in memory as shown below in Fig. 11.2.

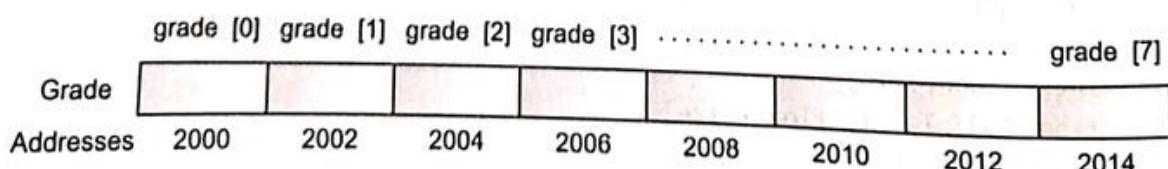


Figure 11.2 An eight-element character array beginning at location 2000.

If you have an *int* array *age* with 5 elements declared as *int age [5]*; its each element will have bytes for its storage. If the starting memory location of array *age* is 5000, then it will be represented in the memory as shown below (in Fig. 11.3).

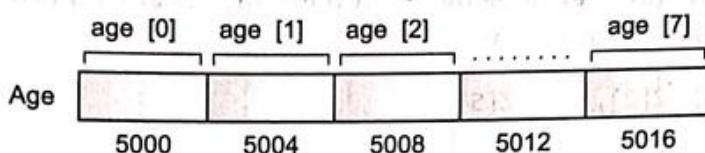


Figure 11.3 Five-element integer array beginning at location 5000.

Internally, arrays are stored as a special object containing :

- ◆ A group of contiguous memory locations that all have the same name and same datatype.
- ◆ A reference that stores the beginning address of the array elements.
- ◆ A separate instance variable containing the number of elements in the array.

Figure 11.4 shows this memory arrangement of arrays.

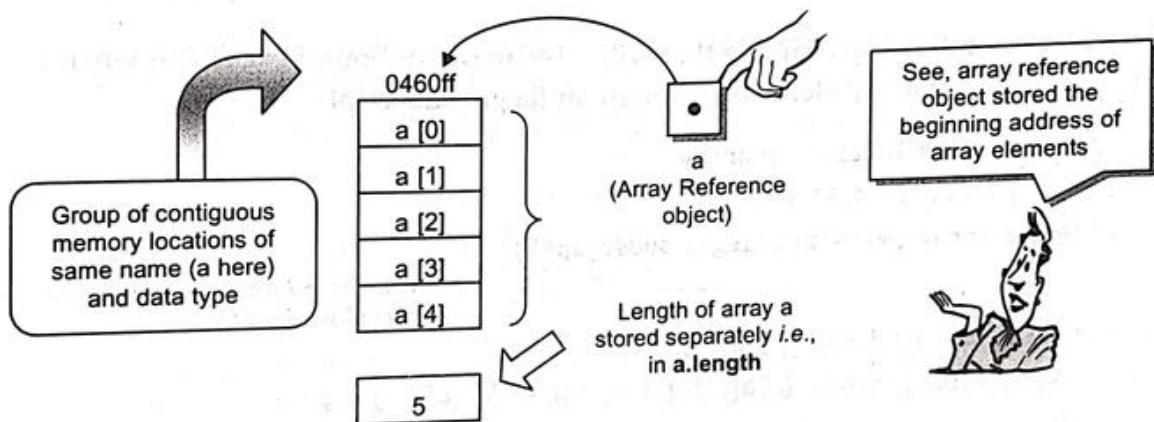


Figure 11.4 Memory arrangement of array objs.

11.3.2 Some Facts about Arrays

Let us now talk about some facts regarding Java arrays. First of all, let us talk about initial values of array elements.

Initializing Arrays

When an array object is created with the *new* operator, its elements are automatically initialized to zero, which is the default initial value for all numeric types. If the array is boolean type then all elements are initialized with boolean false value. Similarly, all elements of character array also get initialized with their default initial value. Do you know what it is ? Well find out.

But you do can initialize arrays to non-zero values using **array initializers**, which are comma-separated list enclosed in braces as shown in following example code. But remember that array initializers *only work in declaration statements*.

```
int a[ ] = {1, 2, 3, 4, 5} ; // Create & initialize array
```



These are array initializers

Out of bound subscripts

Each element of an array is addressed using the name of the array plus the subscripts 0, 1, ..., $n-1$, where n is the number of elements in the array. That is, if array is $Z[6]$ then legal array elements would be referred to as :

$Z[0], Z[1], Z[2], \dots, Z[5]$

Subscripts < 0 or $\geq n$ are illegal, since they do not correspond to real memory locations in the array i.e., for above mentioned array $Z[6]$, following are illegal subscripts (i.e., all other than the legal subscripts, which for this case are 0 - 5) :

$Z[-5], Z[-1], Z[6], Z[7]$ etc.

These subscripts are said to be **out-of-bounds**. Reference to an out-of-bounds subscript produces an **out-of-bounds exception**, and the program will crash if the exception is not handled.

Note The subscripts other than $0 \dots n-1$ (both limits inclusive) for an array having n elements, are called **out-of-bounds subscripts**.

Consider the following example that will raise an out of bound exception when the program tries to access an array elements through an illegal subscript :

```
// Declare and initialize array
int a[ ] = {1,2,3,4,5} ;
// Write array (with an illegal subscript!)
for ( int i = 0 ; i <= 5 ; i++ )
    System.out.println( "a[ " + i + " ] = " + a[ i ] ) ;


Notice when  $i$  will be 5, then  $a[i]$  will refer to an illegal array location
```

The output produced by above code will be as given below :

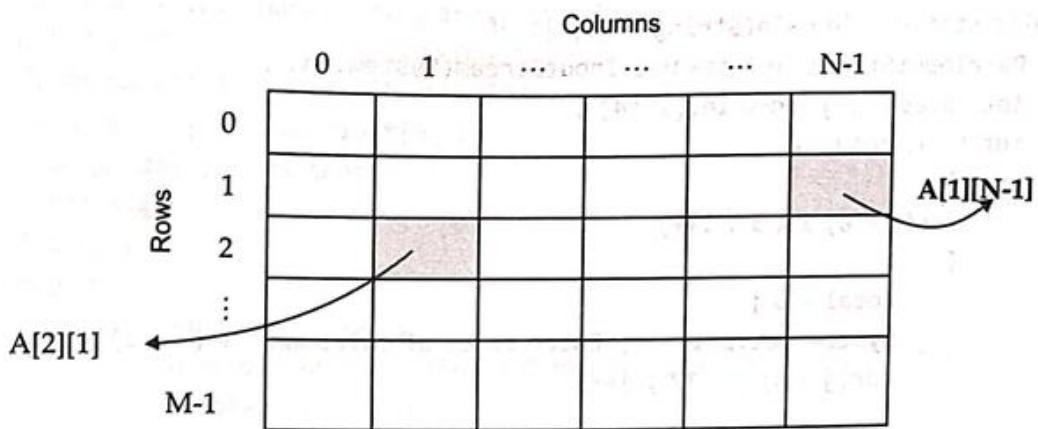
```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
java.lang.ArrayIndexOutOfBoundsException: 5
at TestBounds.main(TestBounds.java:15)
```

Note failure occurs when access to an out-of-bounds subscript i.e., (5) is attempted

11.3.3 Two-Dimensional Arrays

A two-dimensional array is an array in which each element is itself an array, a **1-D array**. For instance, an array $A [M] [N]$ is an M by N table with M rows and N columns containing $M \times N$ elements.

The number of elements in a 2-D array can be determined by multiplying number of rows with number of columns. For example, the number of elements in an array $A [7] [9]$ is calculated as $7 \times 9 = 63$.



The simplest form of a multidimensional array, the two-dimensional array, is an array having single-dimension arrays as its elements. The general form of a two-dimensional array declaration in Java is as follows :

```
type array-name [ ] [ ] = new type [row] [columns];
```

or as `type [] [] array-name = new type[rows] [columns];`

where *type* is the base data type of the array having name *array-name*; *rows*, the first index, refers to the number of rows in the array and *columns*, the second index, refers to the number of columns in the array. Following declaration declares an `int` array *sales* of size 5, 12.

```
int sales [ ] [ ] = new int [5] [12];
```

columns rows

or `int [] [] sales = new int [5] [12];`

The array *sales* have 5 elements *sales* [0], *sales* [1], *sales* [2], *sales* [3] and *sales* [4] each of which is itself an `int` array with 12 elements. The elements of *sales* are referred to as *sales* [0][0], *sales* [0][1],, *sales* [0][11], *sales* [1][0], *sales* [1][1],..... and so forth. The following program (11.4) reads sales of 5 salesmen in 12 months.

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 z[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

```
this is one row of array                           this is another row of array  
int z[ ] [ ] = { {0, 0, 0} , {1, 1, 1} } ;
```

by surrounding the elements of each row by braces.

Program 11.4

Program to read sales of 5 salesmen in 12 months and to print total sales made by each salesman.

```
import java.io.DataInputStream ;
class Salesman {
```

```

public static void main(String args[ ]) {
    DataInputStream in = new DataInputStream(System.in);
    int sales[ ][ ] = new int[2][4];
    int i, j, total;
    try {
        for(i = 0; i < 5; i++)
        {
            total = 0;
            System.out.println("Enter sales of salesman " + (i + 1));
            for(j = 0; j < 12; j++)
            {
                System.out.print("Month " + (j + 1) + " : ");
                sales[i][j] = Integer.parseInt(in.readLine());
                total += sales[i][j];
            }
            System.out.println("Total sales of salesman " + (i + 1) + " = " + total);
        }
    } catch (Exception e) {}
}

```

A sample program run for above program for 2 salesmen's sales in 4 months, is shown below (i.e., we changed condition for *i loop* as $i < 2$ for *j loop* as $j < 4$):

```

Enter sales of salesman 1
Month 1 : 5000
Month 2 : 6000
Month 3 : 7000
Month 4 : 4000
Total sales of salesman 1 = 22000

```

```

Enter sales of salesman 2
Month 1 : 5000
Month 2 : 4000
Month 3 : 4400
Month 4 : 6000
Total sales of salesman 2 = 19400

```

The above program reads for each salesman the monthly sales and simultaneously calculates total sales made by the salesman. But before reading sales of another salesman, it first prints total sales made by the salesmen whose sales have been recently read.

Program 11.5

Program to calculate grades of 5 students from 3 test scores.

```

import java.io.DataInputStream ;
class Student {

```

```

public static void main(String args[ ]) {
    DataInputStream in = new DataInputStream(System.in) ;
    float marks[ ][ ] = new float[5][3];
    float total[ ] = new float[5];
    char grade[ ] = new char[5];
    float avg ;
    int i, j ;
    try {
        for(i = 0; i < 5; i++) {
            System.out.println("Enter Marks for Student " + (i + 1)) ;
            total[ i ] = 0 ;
            for(j = 0; j < 3; j++) {
                System.out.println("Marks in Subject " + (j + 1) + " : ") ;
                marks[ i ][ j ] = Float.parseFloat(in.readLine( )) ;
            }
        }
        for(i = 0; i < 5; i++) {
            total[ i ] = 0 ;
            for(j = 0; j < 3; j++)
                total[ i ] += marks[ i ][ j ] ;
            avg = total[ i ]/3 ;
            if(avg < 45.0)
                grade[ i ] = 'D' ;
            else if(avg < 60.0)
                grade[ i ] = 'C' ;
            else if(avg < 75.0)
                grade[ i ] = 'B' ;
            else
                grade[ i ] = 'A' ;
        }
        for(i = 0; i < 5; i++) {
            System.out.println("Student " + (i + 1)) ;
            System.out.print("Total Marks = " + total[ i ]) ;
            System.out.println("\tGrade = " + grade[ i ]) ;
        }
    } catch (Exception e) { }
}
}

```

Sample Run for 3 Students

```

Enter Marks for Student 1
Marks in Subject 1 :
40
Marks in Subject 2 :
50
Marks in Subject 3 :
40

```

Enter Marks for Student 2

Marks in Subject 1 :

90

Marks in Subject 2 :

80

Marks in Subject 3 :

88

Enter Marks for Student 3

Marks in Subject 1 :

60

Marks in Subject 2 :

50

Marks in Subject 3 :

40

Student 1

Total Marks = 130.0 Grade = D

Student 2

Total Marks = 258.0 Grade = A

Student 3

Total Marks = 150.0 Grade = C

11.3.3A Memory Representation of Two-dimensional Array

Two-dimensional arrays are stored in a row-column matrix, where the first index indicates the row and the second index indicates the column. This means the second index (*i.e.*, *column*) changes faster than the first index (*i.e.*, *row*) when accessing the elements in the array in the order in which they are actually stored in memory. If you have declared an array **pay** as follows :

```
short[ ][ ] pay = new short [5] [7] ;
```

it will be having $5 \times 7 = 35$ elements which will be represented in memory as shown below in Fig. 11.5.

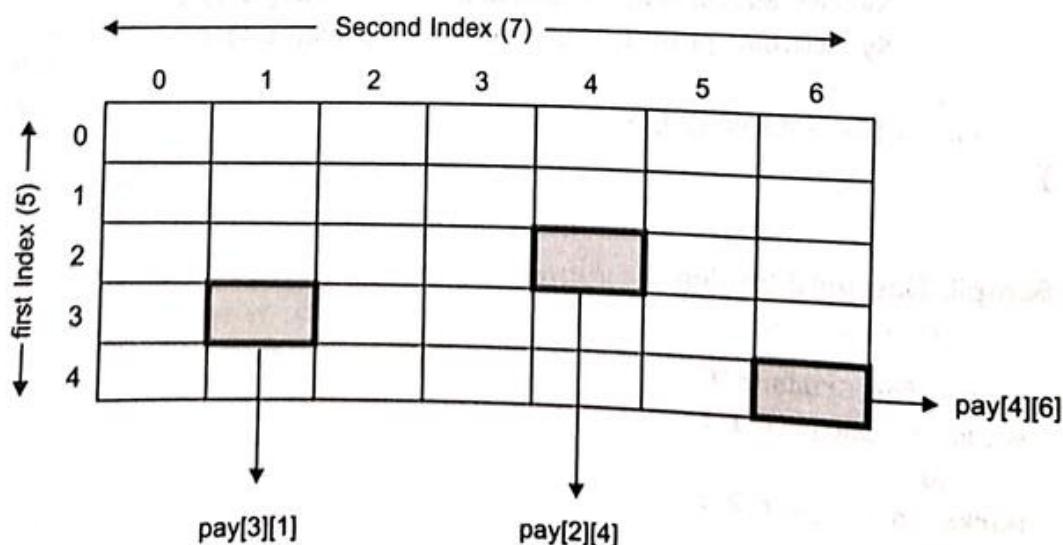


Figure 11.5 A two-dimensional array namely pay [5] [7] in memory.

The amount of storage required to hold a two-dimensional array is also dependent upon its base type, number of rows and number of columns. The formula to calculate total number of bytes required by a two-dimensional array is given below :

$$\text{total bytes} = \text{number-of-rows} \times \text{number of columns} \times \text{size of base type}$$

For instance, the above declared array `pay[5][7]` requires $5 \times 7 \times 2 = 70$ bytes as the byte size is 2 bytes.

We are not going into further details of 2-D arrays, as it is beyond the scope of this book.

11.4 Searching in 1-D Arrays

Sometimes you need to search for an element in an array. To accomplish this task, you can use different searching techniques. Here we are going to discuss two very common search techniques viz., *linear search* and *binary search*.

Linear Search

In linear search, each element of the array is compared with the given *Item* to be searched for, one by one. This method, which traverses the array sequentially to locate the given *Item*, is called *linear search* or *sequential search*.

Program 11.6

Search an element in an array using Linear Search. Pass the element to be searched for, as an argument.

```
class Linear {
    public void lSearch( int n ) {
        int A[ ] = {5, 3, 8, 4, 9, 2, 1, 12, 90, 15};
        int flag = 0, i;
        for(i = 0; i < 10; i++)
        {
            if( n == A[i] )
            {
                flag = 1;
                break;
            }
        }
        if(flag == 1)
            System.out.println("Element present at position" + (i + 1));
        else
            System.out.println("Element not present");
    }
}
```

n is the search-item

As soon as the search-item
is found, the loop is
terminated with a break.

The output produced by above program (for search element $n = 12$) is as shown below :

Element present at position 8

The above program reads an array and takes the item to be searched for (we searched for $n = 12$). It then searches for given item in the given array. If the item is found, then it prints position of found element otherwise it prints "Element n not present in the array".

The above search technique will prove the worst, if the element to be searched is one of the last elements of the array as so many comparisons would take place and the entire process would be time-consuming. To save on time and number of comparisons, binary search is very useful.

Binary Search

This popular search technique searches the given *ITEM* in minimum possible comparisons. The *binary search* requires the array, to be scanned, must be sorted in any order (for instance, say ascending order). In binary search, the *ITEM* is searched for in smaller *segment* (nearly half the previous segment) after every stage. For the first stage, the segment contains the entire array.

To search for *ITEM* in a sorted array (in *ascending order*), the *ITEM* is compared with *middle element* of the segment (*i.e.*, in the entire array for the first time). If the *ITEM* is more than the middle element, latter part of the segment becomes new segment to be scanned ; if the *ITEM* is less than the *middle element*, former part of the segment becomes new segment to be scanned. The same process is repeated for the new segment(s) until either the *ITEM* is found (search successful) or the segment is reduced to the single element and still the *ITEM* is not found (search unsuccessful).

Binary search can work for only sorted arrays whereas linear search can work for both sorted as well as unsorted arrays.

Note Binary Search is a search-technique that works for sorted arrays. Here search-item is compared with the middle element of array. If the search-item matches with the element, search finishes. If search-item is less than middle (in ascending array) perform binary-search in the first half of the array, otherwise perform binary search in the latter half of the array.

Program 11.7

Search an element in an array using Binary Search. Pass the element to be searched for as argument.

```
class Binary
{ public void bSearch(int n)
{
    int A[ ] = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50} ;
    int flag = 0, L, U, M = 0 ;
    L = 0 ;
    U = 9 ;
    while( L <= U )
    {
        M = (L + U)/2 ;
        if( n > A[M] )
            L = M + 1 ;
        else if( n < A[M] )
            U = M - 1 ;
    }
}
```

```

        else
        {
            flag = 1 ;
            break ;
        }
    }
    if(flag == 1)
        System.out.println("Element present at position" + (M + 1)) ;
    else
        System.out.println("Element not present") ;
}
}

```

The output produced by above program (for search element n = 45) is as follows :

Element present at position 9

The above program reads an array AR in ascending order (if its elements are not in ascending order, then the Bsearch() won't work properly), and takes the item to be searched for (we searched for n = 45). It then invokes function Bsearch(), which takes three parameters - *the array, its size, search-item*. The Bsearch() will return the index of found element, in case of successful search. Otherwise, it will return -1.

Example 11.1. Write an algorithm to search for ITEM in an unsorted array INF [-3:5]. Report an unsuccessful search and in case of a successful search interchange ITEM with its previous element.

Solution. As the given array INF is unsorted, linear search is suitable for it.

```

/* Firstly ITEM is to be searched in array INF */
1. ctr = -3
2. Repeat steps 3 and 4 until ctr > 5
3. If INF[ctr] == ITEM then
   { pos = ctr
     break
   }
4. ctr = ctr + 1
/* End of Repeat */
5. if ctr > 5 then
   print "Search Unsuccessful !"
else
if pos != (-3)
{
   temp = INF[pos]           /* Swapping with previous element */
   INF[pos] = INF[pos - 1]
   INF[pos - 1] = temp
}

```

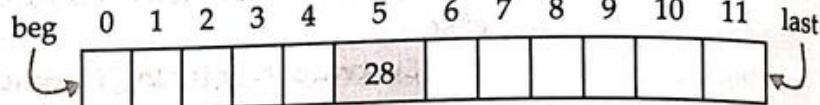
Example 11.2. Write the steps to search 44 and 36 using binary search in the following array DATA :

10	12	14	21	23	28	31	37	42	44	49	53
0	1	2	3	4	5	6	7	8	9	10	11

Solution. (i) Search for 44.

Step I

$$\text{beg} = 0; \quad \text{last} = 11$$



$$\text{mid} = \text{INT} \frac{(0+11)}{2} = \text{Int}(5.5) = 5$$

mid

Step II

Data[mid] i.e., Data[5] is 28

$28 < 44$ then

$$\text{beg} = \text{mid} + 1$$

$$= \text{beg} = 5 + 1 = 6$$

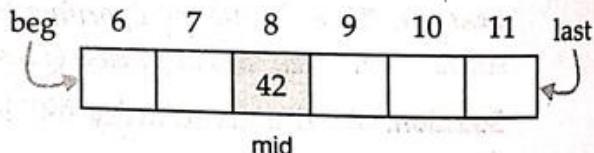
Step III

$$\text{mid} = \text{INT} ((\text{beg} + \text{last})/2) = \text{INT} \left(\frac{(6+11)}{2} = 8 \right)$$

Data[8] i.e., 42 < 44 then

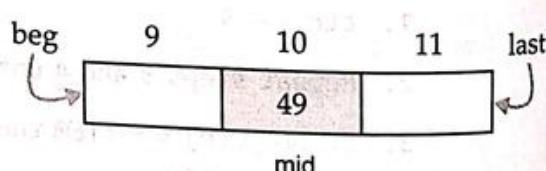
$$\text{beg} = \text{mid} + 1$$

$$\text{beg} = 8 + 1 = 9$$



Step IV

$$\text{mid} = \frac{9+11}{2} = 10$$



Data[10] i.e., 49 > 44 then

$$\text{last} = \text{mid} - 1$$

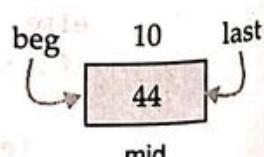
$$\text{last} = 10 - 1 = 9$$

Step V

$$\text{mid} = \text{INT} \left(\frac{(\text{beg} + \text{last})}{2} \right) = \frac{(9+9)}{2} = 9$$

(beg = last = 8)

Data [9] i.e., 44 = 44



Search Successful !! At index 9.

(ii) Search for 36

Step I

$$\text{beg} = 0; \text{last} = 11$$

$$\text{mid} = \text{INT}\left(\frac{10+11}{2}\right) = 5$$

Step III

$$\text{mid} = \text{INT}\left(\frac{6+11}{2}\right) = 8$$

Data[8] i.e., 42

42 > 36 then

$$\text{last} = \text{mid} - 1 = 8 - 1 = 7$$

Step II

Data [mid] i.e., Data [5] is 28

28 < 36 then

$$\text{beg} = \text{mid} + 1 = 5 + 1 = 6$$

Step IV

$$\text{mid} = \left(\frac{6+7}{2}\right) = 6$$

Data [6] is 31

$$31 < 36 \text{ then } \text{beg} = \text{mid} + 1 = 6 + 1 = 7$$

Step V

$$\text{mid} = \left(\frac{7+7}{2}\right) = 7 \quad (\text{beg} = \text{last} = 7)$$

Data [7] is 37

37 ≠ 36

Search Unsuccessful !!**Example 11.3.** Write the steps to search 34.5 using binary search in the following array Amount [7] :

60.4	53.22	41.05	41.00	39.20	34.5	21.15
0	1	2	3	4	5	6

Solution. The given array is in descending order ; ITEM to be searched for is 34.5.**Step I**

$$\text{beg} = 0; \text{last} = 6$$

$$\text{mid} = \text{INT}((\text{beg} + \text{last}) / 2)$$

$$= \text{INT}((0 + 6) / 2)$$

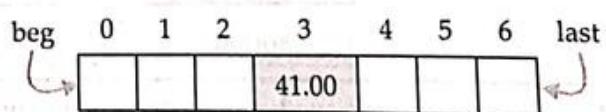
$$= \text{INT}(3) = 3$$

Step II

Amount[mid] i.e., Amount[3] is 41

41 > 34.5 then

$$\text{beg} = \text{mid} + 1 = 3 + 1 = 4$$

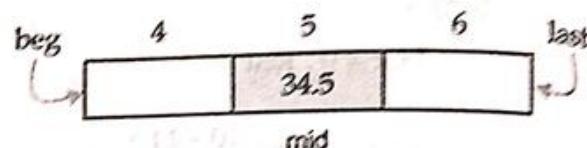


Step III

$$\begin{aligned} \text{mid} &= \text{INT} ((\text{beg} + \text{last}) / 2) \\ &= \text{INT} ((4 + 6) / 2) = \text{INT} (5) = 5. \end{aligned}$$

Data[mid] i.e., Data[5] is 34.5

$$34.5 = 34.5$$



Search Successful !! At index 5. (position 6)

11.5 Sorting

Sorting of an array means arranging the array elements in a specified order i.e., either ascending or descending order. There are several sorting techniques available e.g., shell sort, shuttle sort, bubble sort, selection sort, quick sort, heap sort etc. But we will be covering only two very common and popular techniques¹ here viz., selection sort and bubble sort.

Selection Sort

One family of internal sorting algorithms is the group of *selection sorts*. The basic idea of a selection sort is to repeatedly select the smallest key in the remaining unsorted array.

For example, consider the following unsorted array to be sorted using selection sort

15	6	13	22	3	52	
unsorted array						

Step I. Choose the smallest element from unsorted array which is 2 ; put this smallest element in sorted part at 1st position. Now array becomes as follows :

2	15	6	13	22		3	52	
sorted		unsorted array						

Step II. The second pass identifies 3 as the smallest element in remaining unsorted array. Adding 3 also in sorted part at 2nd position we have the following array :

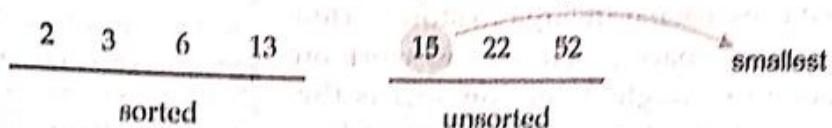
2	3	15		6	13	22	52	
sorted		unsorted						

Step III. In third pass, 6 is chosen as smallest and put at position 3 and our array becomes

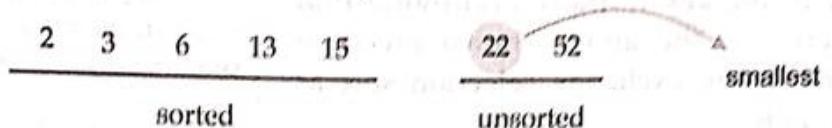
2	3	6	15		13	22	52	
sorted			unsorted					

1. Appendix A covers another popular sorting technique *Insertion Sort*.

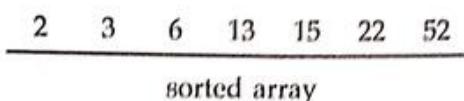
Step IV. This time 13 is moved to sorted part at 4th position



Step V. Fifth pass adds 15 to sorted part at the next position.

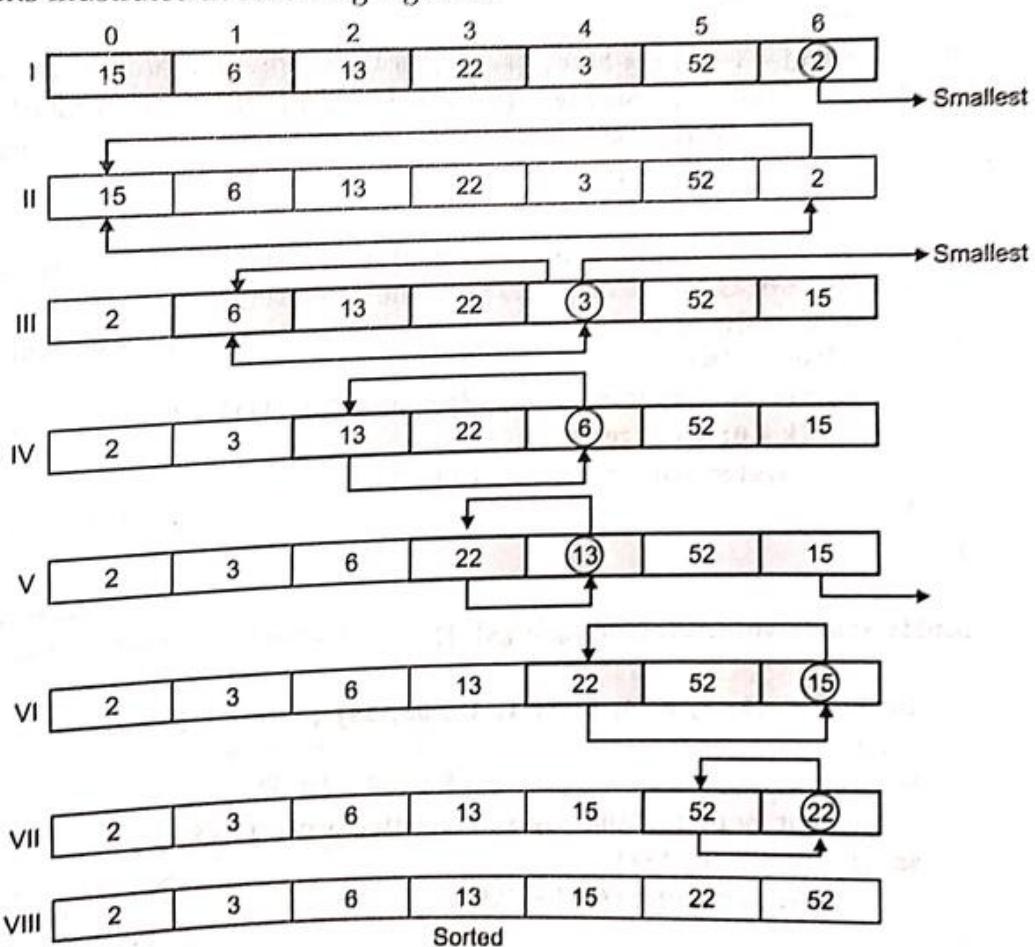


Step VI. Sixth pass adds 22 and seventh pass adds 52 to sorted list



Step VII. Therefore after seven passes the array is sorted.

On our machines, selection sort is implemented in the form of *exchange selection sort* that requires a single array to work with. In this technique, computer keeps on finding the next smallest element (the smallest of all, for the very first time) and brings it at its appropriate positions illustrated in following Fig. 11.6.



○ Depicts smallest element in unsorted part. □ Surrounds unsorted part of the array.

Figure 11.6 Exchange selection sort.

The above mentioned first *selection sort* technique (in seven steps) uses a separate area to store sorted part, therefore, more space is required to work on it. A modification to straight selection sort is the *exchange selection sort* wherein the smallest key is moved into its final position by being exchanged with the key initially occupying that position. Therefore, the above shown unsorted array is sorted, using exchange selection sort as shown in Fig. 11.6.

Note SelectionSort is a sorting technique where next smallest (or next largest) element is found in the array and moved to its correct position (find position) e.g., the smallest element should be at 1st position (for ascending array), second smallest should be at 2nd position and so on.

Program 11.8

Sort an array using Exchange Selection Sort.

```

class Selection {
    public static void Selsort(int A[ ], int size)
    {
        int i, j, k, small, tmp, pos ;
        for(i = 0; i < size; i++) {
            small = A[ i ] ;
            pos = i ;
            for(j = i + 1; j < size; j++) {
                if(A[ j ] < small) {
                    small = A[ j ] ;
                    pos = j ;
                }
            }
            tmp = A[i] ;
            A[i] = A[pos] ;
            A[pos] = tmp ;
            System.out.println("Array after pass " + (i+1) + " is -> ") ;
            for(k = 0; k < size; k++)
                System.out.print(A[k] + "\t") ;
        }
    }

    public static void main(String args[ ])
    {
        int A[ ] = {5, 3, 8, 4, 9, 2, 1, 12, 90, 15} ;
        int i;
        Selsort(A,10);
        System.out.println("\nArray in ascending order is -> ") ;
        for(i = 0; i < 10; i++)
            System.out.print(A[i] + "\t") ;
    }
}

```

```

Array after pass 1 is ->
1   3   8   4   9   2   5   12   90
Array after pass 2 is ->
1   2   8   4   9   3   5   12   90
Array after pass 3 is ->
1   2   3   4   9   8   5   12   90
Array after pass 4 is ->
1   2   3   4   9   8   5   12   90
Array after pass 5 is ->
1   2   3   4   5   8   9   12   90
Array after pass 6 is ->
1   2   3   4   5   8   9   12   90
Array after pass 7 is ->
1   2   3   4   5   8   9   12   90
Array after pass 8 is ->
1   2   3   4   5   8   9   12   90
Array after pass 9 is ->
1   2   3   4   5   8   9   12   90
Array after pass 10 is ->
1   2   3   4   5   8   9   12   15
Array in ascending order is ->
1   2   3   4   5   8   9   12   15

```

The above program not only sorts the given array, but also shows you array after every pass. To sort the given array, this program calls function `Selsort()`, which performs selection sort on the passed array.

Bubble Sort

The basic idea of bubble sort is to compare two adjoining values and exchange them if they are not in proper order. For instance, following unsorted array is to be sorted in ascending order using bubble sort. In every pass, the heaviest element settles as its appropriate position in the bottom.

Program 11.9

Sort an array using Bubble Sort.

```

class Bubble {
    public static void Bubblesort(int A[], int size) {
        int i, j, k, tmp ;
        for(i = 0; i < size; i++) {
            for(j = 0; j < size - 1 - i; j++) {
                if (A[j] > A[j + 1]) {
                    tmp = A[j] ; A[j] = A[j + 1] ; A[j + 1] = tmp ;
                }
            }
        }
    }
}

```

Two adjacent elements
are being compared

```

        System.out.println("Array after pass " + (i+1) + " is -> ");
        for(k = 0; k < size; k++) System.out.print(A[k] + "\t");
    }
}

public static void main(String args[ ])
{
    int A[ ] = {15, 3, 8, 4, 9, 2, 5, 12, 90, 1} ;
    Bubblesort(A,10);
    System.out.println("\nArray in ascending order is -> ");
    for(int i = 0; i < 10; i++) System.out.print(A[i] + "\t");
}
}

```

Array after pass 1 is ->

3	8	4	9	2	5	12	15	1	90
---	---	---	---	---	---	----	----	---	----

Array after pass 2 is ->

3	4	8	2	5	9	12	1	15	90
---	---	---	---	---	---	----	---	----	----

Array after pass 3 is ->

3	4	2	5	8	9	1	12	15	90
---	---	---	---	---	---	---	----	----	----

Array after pass 4 is ->

3	2	4	5	8	1	9	12	15	90
---	---	---	---	---	---	---	----	----	----

Array after pass 5 is ->

2	3	4	5	1	8	9	12	15	90
---	---	---	---	---	---	---	----	----	----

Array after pass 6 is ->

2	3	4	1	5	8	9	12	15	90
---	---	---	---	---	---	---	----	----	----

Array after pass 7 is ->

2	3	1	4	5	8	9	12	15	90
---	---	---	---	---	---	---	----	----	----

Array after pass 8 is ->

2	1	3	4	5	8	9	12	15	90
---	---	---	---	---	---	---	----	----	----

Array after pass 9 is ->

1	2	3	4	5	8	9	12	15	90
---	---	---	---	---	---	---	----	----	----

Array after pass 10 is ->

1	2	3	4	5	8	9	12	15	90
---	---	---	---	---	---	---	----	----	----

Array in ascending order is ->

1	2	3	4	5	8	9	12	15	90
---	---	---	---	---	---	---	----	----	----

Array in ascending order is ->

1	2	3	4	5	8	9	12	15	90
---	---	---	---	---	---	---	----	----	----

The above program reads an array and then invokes function **Bubblesort()** to sort this array. The above program also shows you the array after each iteration.

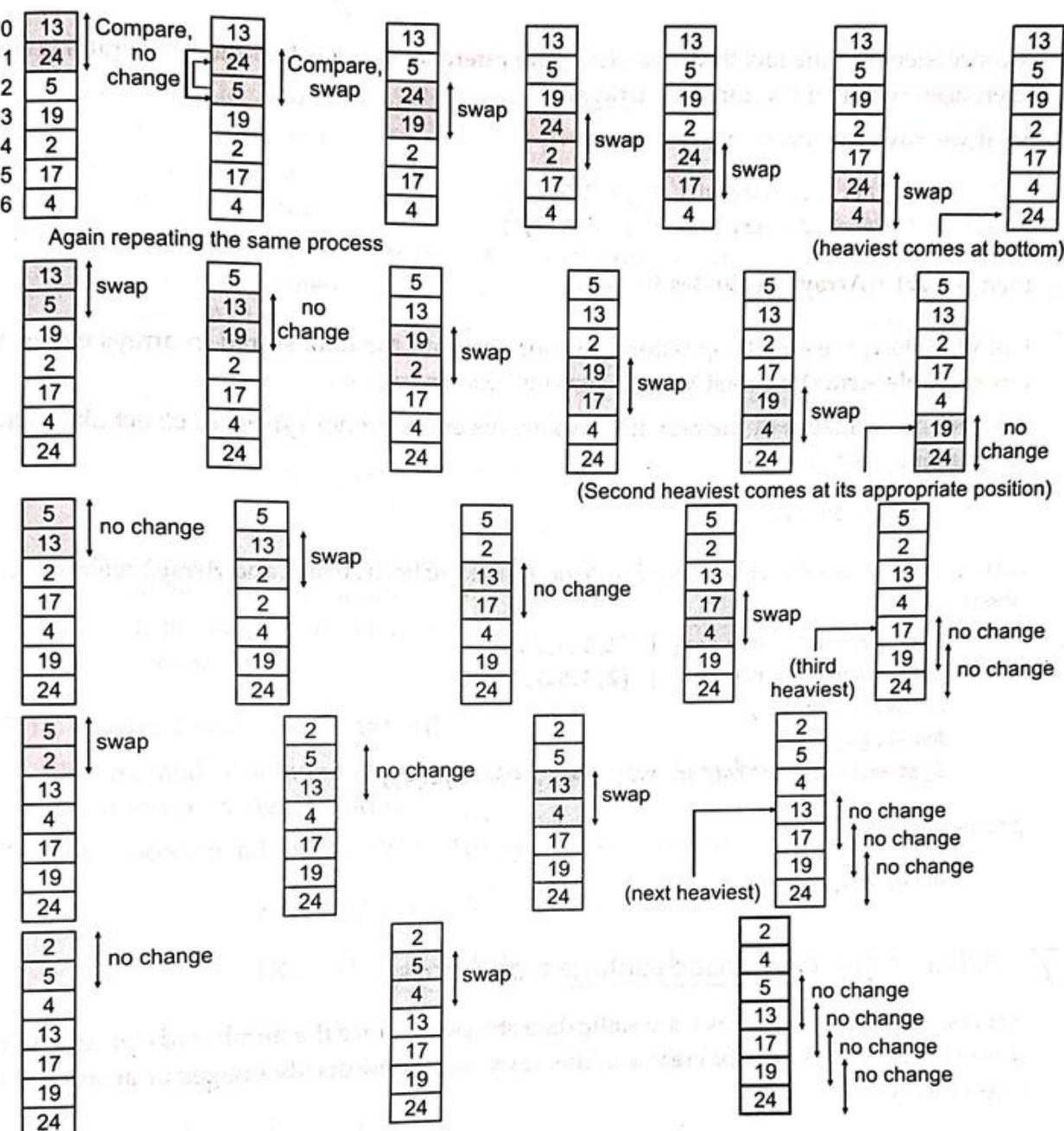


Figure 11.7 Bubble sort

11.6 Arrays vs. Objects

Arrays are treated as objects as Java arrays are reference types. For example, consider this :

The following code

```
T[ ] a = new T[n]
```

defines an array variable with name *a* whose components are of type *T* and allocates memory space for *n* such components as shown in Fig. 11.8.

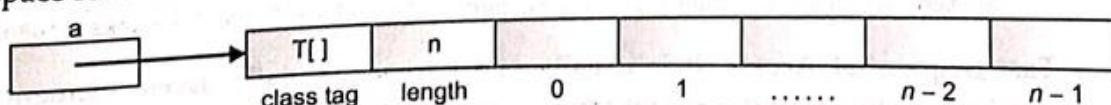


Figure 11.8

A consequence of the fact that Java arrays are reference types is that the == operator compares references not the data stored in arrays.

So, if we have the following two arrays :

```
int[ ] Array1 = new int[ ] {2,3,5,7} ;
int[ ] Array2 = new int[ ] {2,3,5,7} ;
```

then Array1==Array2 evaluates to false.

Unfortunately, the equals operator does not compare the data stored in arrays either. So, in our example Array1.equals(Array2) again evaluates to false.

Another consequence of the fact that Java arrays are reference types i.e., object like, is that an assignment statement like

```
Array1 = Array2
```

copies the address stored in Array2 to Array1, making both Array1 and Array2 refer to the same object

```
int[ ] Array1 = new int[ ] {2,3,5,7} ;
int[ ] Array2 = new int[ ] {2,3,5,7,11} ;
Array1 = Array2 ;
Array2[4] = 13 ;
System.out.println("Array1[4] = " + Array1[4]) ;
```

prints out

Array1[4] = 13

11.7 Advantages and Disadvantages of Arrays

Arrays, including Java arrays, are static data structures since the number of elements they can store is fixed at the time of creation. The advantages and disadvantages of arrays are being listed below :

Advantages

- (i) **Easy to specify.** The declaration, allocation of memory space, initialization can all be done in one line of code.
- (ii) **Free from run-time overheads.** There is no run-time overhead to allocate/free memory, apart from once at the start and end.
- (iii) **Random access of elements.** Arrays facilitate random (direct) access to any element via its index or subscript e.g., direct access to a student record via student number (index can be derived from student number) e.g.,

```
byte stuRollNo = 5 ;
```

```
System.out.println("Marks of roll number "+stuRollNo+ " are " + marks[stuRollNo] ) ;
```

- (iv) **Fast Sequential Access.** It is usually faster to sequentially access elements due to contiguous storage and constant time computation of the address of a component.

Disadvantages

- (i) **Need to know the size.** A potential disadvantage of arrays (depending on the application) is the need to know the size at the time of allocation.
- (ii) **Careful design required.** Careful design is required to make sure that the array will be large enough to hold the largest possible group of data but no larger.
- (iii) **Not suitable for situations demanding varying memory-sizes.** For situations wherein the memory requirements are not known before hand, arrays are not suitable at all. As here, arrays may either lead to shortage-of-memory or wastage-of-memory

11.8 Solution of Simultaneous Linear Equations

Reliable subroutines for the solution of simultaneous linear equations of the form

$$AX = B$$

are available in the program libraries of most computer installations. One of the simplest methods for solving simultaneous equations is Gauss method which is more exact among all the methods.

11.8.1 Gauss Elimination Method

This method of solution eliminates the variables and finally the set of equations is reduced into a lower triangular form.

The procedure adopted is illustrated by the following example :

$$5X_1 + 3X_2 - X_3 = 3$$

$$2X_1 + 5X_2 + X_3 = -5$$

$$2X_1 + X_2 + 2X_3 = 4$$

The set of the equations should be written with $a_{11} = 1$

$$\therefore X_1 + \frac{3}{5}X_2 - \frac{1}{5}X_3 = \frac{3}{5}$$

$$2X_1 + 5X_2 + X_3 = -5 \left(r_{12} = \frac{a_{21}}{a_{11}} = \frac{2}{1} = 2 \right)$$

$$2X_1 + X_2 + 2X_3 = 4 \left(r_{13} = \frac{a_{31}}{a_{11}} = \frac{2}{1} = 2 \right)$$

In arranging the above equations the required condition is $a_{11} \neq 0$. Then X_1 is eliminated from the second through n^{th} equation by defining the multipliers

$$r_{1i} = \frac{a_{1i}}{a_{11}}, \quad i = 2, 3, \dots, n$$

and by subtracting r_{1i} times the first equation from the i^{th} equation.

First attempt

$$X_1 + \frac{3}{5}X_2 - \frac{1}{5}X_3 = \frac{3}{5}$$

$$+ 19X_2 + 7X_3 = -31$$

$$- X_2 + 12X_3 = 14$$

$$(a) \quad 2X_1 + 5X_2 + 1X_3 = -5$$

$$2X_1 + \frac{6}{5}X_2 - \frac{2}{5}X_3 = \frac{6}{5}$$

$$\underline{- \quad - \quad + \quad -}$$

$$\frac{19}{5}X_2 + \frac{7}{5}X_3 = -\frac{31}{5}$$

$$\therefore 19X_2 + 7X_3 = -31$$

$$(b) \quad 2X_1 + 1X_2 + 2X_3 = 4$$

$$2X_1 + \frac{6}{5}X_2 - \frac{2}{5}X_3 = \frac{6}{5}$$

$$\underline{- \quad - \quad + \quad -}$$

$$-\frac{1}{5}X_2 + \frac{12}{5}X_3 = \frac{14}{5}$$

$$\therefore -X_2 + 12X_3 = 14$$

The set of equations now should be written as $a_{22} = 1$

$$\therefore X_1 + \frac{3}{5}X_2 - \frac{1}{5}X_3 = \frac{3}{5}$$

$$X_2 + \frac{7}{19}X_3 = -\frac{31}{19} \quad \text{where } r_{22} = \frac{-1}{1} = -1$$

$$-X_2 + 12X_3 = 14$$

Second attempt

$$r_{2i} = \frac{a_{i2}}{a_{22}} \quad \text{where } i = 3, 4, \dots, n$$

$$X_1 + \frac{3}{5}X_2 - \frac{2}{5}X_3 = \frac{3}{5}$$

$$X_2 + \frac{7}{19}X_3 = -\frac{31}{19}$$

$$X_2 = 1$$

$$(a) \quad -X_2 + 12X_3 = 14$$

$$-X_2 - \frac{7}{19}X_3 = \frac{31}{19}$$

$$\underline{+ \quad + \quad -}$$

$$\frac{235}{19}X_3 = \frac{235}{19}$$

$$\therefore X_3 = 1$$

The original set of equations can be written in matrix form as

$$\begin{bmatrix} 5 & 3 & -1 & : & 3 \\ 2 & 5 & 1 & : & -5 \\ 2 & 1 & 2 & : & 4 \end{bmatrix}$$

This form is reduced to

$$\left[\begin{array}{ccc|c} 1 & \frac{3}{5} & -\frac{1}{5} & \frac{3}{5} \\ 0 & 1 & \frac{7}{19} & -\frac{31}{19} \\ 0 & 0 & 1 & 1 \end{array} \right]$$

This set of 3 equations is same as the original set but the matrix is changed to an upper triangular matrix. The advantage of this is, the values in the last row directly give the value of the one variable.

$$\therefore \frac{235}{19} X_3 = \frac{235}{19} \quad \therefore X_3 = 1$$

$$X_2 + \frac{7}{19} X_3 = -\frac{31}{19}$$

$$\therefore X_2 = -\frac{31}{19} - \frac{7}{19} \times 1 = -\frac{38}{19} = -2$$

$$X_1 + \frac{3}{5} X_2 - \frac{1}{5} X_3 = \frac{3}{5}$$

$$\therefore X_1 = \frac{3}{5} - \frac{3}{5}(-2) + \frac{1}{5} \times 1 = \frac{10}{5} = 2$$

$$\therefore X_1 = 2, X_2 = -2 \text{ and } X_3 = 1.$$

Example 11.4. Solve the following set of equations by using Gauss elimination method

$$6X + 3Y + 2Z = 1$$

$$6X + 4Y + 3Z = 0$$

$$20X + 15Y + 12Z = 0$$

Solution. The X is eliminated from the second and third equations by defining the multipliers

$$r_{1i} = \frac{a_{i1}}{a_{11}}, \quad i = 2, 3, \dots, n$$

and by subtracting r_{1i} times the first equation from second and third.

First attempt

$$6X + 3Y + 2Z = 1$$

$$6X + 4Y + 3Z = 0 \left(r_{12} = \frac{a_{21}}{a_{11}} = \frac{6}{6} = 1 \right)$$

$$20X + 15Y + 12Z = 0 \quad \left(r_{13} = \frac{a_{31}}{a_{11}} = \frac{20}{6} = \frac{10}{3} \right)$$

$$6X + 3Y + 2Z = 1$$

$$Y + Z = -1$$

$$(a) \quad 6X + 4Y + 3Z = 0$$

$$6X + 3Y + 2Z = 1$$

$$\underline{\underline{- \quad - \quad -}}$$

$$Y + Z = -1$$

$$5Y + \frac{16}{3}Z = -\frac{10}{3} \quad \left(r_{23} = \frac{a_{32}}{a_{22}} = \frac{5}{1} = 5 \right)$$

$$(b) \quad 20X + 15Y + 12Z = 0$$

$$20X + 10Y + \frac{20}{3}Z = \frac{10}{3}$$

$$\underline{\underline{- \quad - \quad -}}$$

$$5Y + \frac{16}{3}Z = -\frac{10}{3}$$

Second attempt

The Y is eliminated from third equation by using a multiplier

$r_{23} = \frac{a_{32}}{a_{22}}$ and subtracting r_{23} times the second equation from third.

$$6X + 3Y + 2Z = 1$$

$$Y + Z = -1$$

$$5Y + \frac{16}{3}Z = -\frac{10}{3}$$

$$\frac{1}{3}Z = \frac{5}{3}$$

$$5Y + 5Z = -5$$

$$\underline{\underline{- \quad - \quad +}}$$

$$\frac{1}{3}Z = \frac{5}{3}$$

The original matrix

$$\begin{bmatrix} 6 & 3 & 2 & : & 1 & 12 \\ 6 & 4 & 3 & : & 0 & 13 \\ 20 & 15 & 12 & : & 0 & 47 \end{bmatrix}$$

is changed to the following after using Gaussian elimination method as

$$\begin{bmatrix} 6 & 3 & 2 & : & 1 & 12 \\ 0 & 1 & 1 & : & -1 & 1 \\ 0 & 0 & 1/3 & : & 5/2 & 2 \end{bmatrix}$$

The last column in (a) and (b) is the sum of all the elements along the rows. This keeps the check on the calculations during the execution of the program.

Now the values of X, Y, Z are calculated as

$$\frac{1}{3}Z = \frac{5}{3} \quad \therefore \quad Z = 5$$

$$Y + Z = -1$$

$$\therefore Y = -1 - Z = -6$$

$$6X + 3Y + 2Z = 1$$

$$\therefore 6X = 1 - 3Y - 2Z = 1 + 18 - 10 = 9$$

$$\therefore X = \frac{9}{6} = 1.5$$

$$\therefore X = 1.5, Y = -6 \text{ and } Z = 5.$$

Note Diagonal term need not be one for using this method but it must form upper diagonal matrix.

11.8.2 Generalised Solution by Gauss Elimination Method and Corresponding Computer Program

The Gauss elimination procedure for solving a set of simultaneous equations of the form

$$AX = B$$

consists of two major passes. In the forward pass, the equations are reduced to an upper triangular form, that is, a new matrix A' is generated such that

$$a_{ij}' = 0 \text{ if } j < i$$

This process is performed by a symmetric elimination of lower triangular elements and the vector B is modified accordingly. In the backward pass, the unknowns are obtained in sequence in terms of previously computed unknowns. Consider a set of equations as

$$a_{11} X_1 + a_{12} X_2 + \dots + a_{1n} X_n = b_1$$

$$a_{21} X_1 + a_{22} X_2 + \dots + a_{2n} X_n = b_2$$

.....

$$a_{n1} X_1 + a_{n2} X_2 + \dots + a_{nn} X_n = b_n$$

where a 's and b 's are known constants and X 's are unknowns to be calculated.

The above set of equations' constants can be represented in matrix form as

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{bmatrix}$$

including the constants on the right-hand side of the above set of equations. The matrix obtained now is known as *augmented matrix*.

Each row of this matrix represents one equation and only operation which can be performed with this equation can also be performed with the above matrix.

The following procedure is outlined for solving the above set of equations with an assumption that the equations are so ordered that $a_{11} \neq 0$.

For convenience the elements in the last column of the matrix are represented by

$$a_{i,n+1} \text{ where } i=1, 2, \dots, n$$

The matrix now can be represented as

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & a_{1,n+1} \\ a_{21} & a_{22} & \dots & a_{2n} & a_{2,n+1} \\ \vdots & \vdots & \dots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & a_{n,n+1} \end{bmatrix}$$

(a) The first step to start with is to divide each element in the first row a_{1j} by the first element a_{11} in that row

$$a'_{1j} = \frac{a_{1j}}{a_{11}}$$

Then the matrix becomes

$$\begin{bmatrix} 1 & a'_{12} & a'_{13} & \dots & a'_{1n} & a'_{1,n+1} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} & a_{2,n+1} \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} & a_{n,n+1} \end{bmatrix}$$

The first equation is now multiplied by the first element of each other row in turn and it (first equation) is subtracted from that row. Thus the constant of the first variable K_1 becomes zero in every row except the first.

$$(b) a_{ij} = a_{ij} - a_{ik} a_{ik}, \text{ where } j=k \text{ to } n+1 \text{ and}$$

$$i=k+1 \text{ to } n \text{ using old values of } a_{ik}.$$

The last equation in matrix (1) has only one unknown in it, and it can be easily found.

$$X_n = a''_{n,n+1} / a''_{nn}$$

Now, if we proceed back, it is obvious that each preceding equation has one more unknown than the one below it.

The last but one equation is

$$X_{n-1} + a''_{n-1,n} X_n = a''_{n-1,n+1}$$

and its solution is $X_{n-1} = a''_{n-1,n+1} - a''_{n-1,n} X_n$

as the value of X_n is known from the previous equation.

Therefore, in general K^{th} equation can be solved by

$$X_k = a''_{k,n+1} - \left[\sum_{j=k+1}^n (a''_{kj} X_j) \right] \text{ for } k = (n-1) \text{ to } 1$$

All the operations mentioned above can be represented algebraically as follows in continuation to (b)

(c) $k = k + 1$ and go back to (b) if $k < n$

(d) $X_n = a_{n,n+2} / a_{nn}$

(e) $k = n - 1$

$$(f) X_k = a_{k,n+1} - \sum_{j=k+1}^n (a_{kj} X_j)$$

(g) $k = k - 1$ and go back to (g) if $k > 0$

(h) when $k = 0$, all the X 's are calculated.

Total number of divisions and multiplications required in this process are n and $n(n+1)(2n+1)/6$ respectively. A set of 10 equations requires only 385 multiplications

$$a'_{ij} = a_{ij} - a'_{1j} a_{i1}, \quad \text{where } i = 2, 3, \dots, n \quad \text{and } j = 1, 2, 3, \dots, (n+1)$$

Hence for $j=1$ (first column) $a'_{i1} = a_{i1} - a'_{11} a_{i1}$

$$a'_{11} = 1 \quad \therefore \quad a'_{i1} = a_{i1} - a_{i1} = 0$$

This states that all the elements except first in the first row become zero in the first column and the matrix now becomes

$$\begin{bmatrix} 1 & a'_{12} & a'_{13} & \dots & a'_{1n} & a'_{1,n+1} \\ 0 & a'_{22} & a'_{23} & \dots & a'_{2n} & a'_{2,n+1} \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & a'_{n2} & a'_{n3} & \dots & a'_{nn} & a'_{n,n+1} \end{bmatrix}$$

Now we ignore the first column and the first row and repeat this process on $(n-1)$ equations in $(n-1)$ unknowns. By repeating this process $(n-1)$ times, the following matrix is obtained.

$$\begin{bmatrix} 1 & a''_{12} & a''_{13} & \dots & a''_{1n} & a''_{1,n+1} \\ 0 & 1 & a''_{23} & \dots & a''_{2n} & a''_{2,n+1} \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a''_{nn} & a''_{n,n+1} \end{bmatrix} \quad \dots(1)$$

This set of n equations in n unknowns is the same as the original set but the matrix has been changed to an upper triangular matrix. The steps followed to achieve this can be summarised as

(a) $k = 1$ (equation number)

(b) $a_{kj} = a_{kj} / a_{kk}$, where $j = k$ to $n+1$ using old values of a_{kk}

The program to Gauss Elimination method is as follows :

Program 11.10

Program to solve linear equations through Gauss Elimination method.

```
//Gauss Elimination Method to solve linear equations
import java.io.*;
import java.lang.*;
class GaussSol {
    public static void main(String args[ ]) {
        int n, i, j ;
        DataInputStream in = new DataInputStream(System.in) ;
        try {
            System.out.println("What is the size of the system (n) <10 : ");
            n = Integer.parseInt(in.readLine( )) ;
            double x[ ] = new double[n];
            double b[ ] = new double[n];
            double a[ ][ ] = new double[n][n] ;
            System.out.println("Enter coefficients a(i, j), row-wise -> ");
            for(i=0; i<n; i++) {
                System.out.println("Row " + (i+1) );
                for(j=0 ; j < n ; j++)
                    a[ i ][ j ] = Double.valueOf(in.readLine()).floatValue( ) ;
            }
            System.out.println("Input vector b -> ");
            for(j = 0 ; j< n ; j++)
                b[j] = Double.valueOf(in.readLine( )).floatValue( ) ;
            int index[ ] = new int[n];
            x = Gaussian(a, b, index);      // solve linear equations
            System.out.println( "Solution vector X " );
            for (i=0; i<n; i++)
                System.out.print( x[i] + " " );
        }
        catch (Exception e) {
            System.out.print("Exceptional Error occurred! " );
        }
    }

    // Method to solve the eqn a[ ][ ] x[ ] = b[ ] with the partial-pivoting Gaussian elimination.
    public static double[ ] Gaussian (double a[ ][ ], double b[ ], int index[ ]) {
        int n = b.length;
        double x[ ] = new double[n];
        // Transform the matrix into an upper triangle
        gaussian(a, index);
        // Update the array b[i] with the ratios stored
        for(int i=0; i<n-1; ++i) {
```

```
for(int j = i+1; j < n; ++j) {  
    b[ index[ j ] ] -= a[ index[ j ] ][ i ] * b[ index[ i ] ];  
}  
}  
  
// Perform backward substitutions  
x[ n-1 ] = b[ index[ n-1 ] ] / a[ index[ n-1 ] ][ n-1 ];  
for (int i= n - 2 ; i >= 0; --i) {  
    x[ i ] = b[ index[ i ] ];  
    for (int j = i +1; j < n; ++j) {  
        x[ i ] -= a[ index[ i ] ][ j ] * x[ j ];  
    }  
    x[ i ] /= a[ index[ i ] ][ i ];  
}  
  
// Put x[ i ] into the correct order  
order( x, index );  
return x;  
}  
  
// Method to carry out the partial-pivoting Gaussian elimination.  
// index[ ] stores pivoting order.  
public static void gaussian( double a[ ][ ], int index[ ] ) {  
    int n = index.length;  
    double c[ ] = new double[ n ];  
  
    // Initialize the index  
    for (int i=0; i < n; ++i)  
        index[ i ] = i;  
  
    // Find the rescaling factors, one from each row  
    for (int i=0; i < n; ++i) {  
        double c1 = 0;  
        for (int j = 0; j < n; ++j) {  
            double c0 = Math.abs( a[ i ][ j ] );  
            if (c0 > c1)  
                c1 = c0;  
        }  
        c[ i ] = c1;  
    }  
    // Search the pivoting element from each column  
    int k = 0;  
    for (int j=0; j < n-1; ++j) {  
        double pi1 = 0;  
        for (int i=j; i < n; ++i) {  
            double pi0 = Math.abs( a[ index[ i ] ][ j ] );  
            pi0 /= c[ index[ i ] ];  
            if (pi0 > pi1) {  
                k = i;  
            }  
        }  
        if (k != j) {  
            swapRow( a, index, j, k );  
            swapRow( c, index, j, k );  
        }  
    }  
}
```

```

        pi1 = pi0;
        k = i;
    }

}

// Interchange rows according to the pivoting order
int itmp = index[ j ];
index[ j ] = index[ k ];
index[ k ] = itmp;
for (int i=j+1; i<n; ++i) {
    double pj = a[ index[i] ][ j ] / a[ index[j] ][ j ];
    // Record pivoting ratios below the diagonal
    a[ index[ i ] ][ j ] = pj;
    // Modify other elements accordingly
    for (int l=j+1; l<n; ++l)
        a[ index[ i ] ][ l ] -= pj*a[ index[ j ] ][ l ];
}
}

// Method to sort an array x[i] from the lowest to the highest value of index[i].
public static void order(double x[ ], int index[ ])
{
    int m = x.length;
    for (int i = 0; i<m; ++i)
    {
        for (int j = i+1; j<m; ++j) {
            if (index[ i ] > index[ j ] )
            {
                int itmp = index[ i ];
                index[ i ] = index[ j ];
                index[ j ] = itmp;
                double xtmp = x[ i ];
                x[ i ] = x[ j ];
                x[ j ] = xtmp;
            }
        }
    }
}

```

The output produced is

What is the size of the system (n) <10 :

3

Enter coefficients a(i, j), row-wise ->

Row 1

2

2

1

Row 2

4

2

3

Row 3

1

1

1

Input vector b ->

6

4

0

Solution vector x

5.0

1.0

- 6.0

Let Us Revise

- ❖ An array is a collection of variables of the same type that are referenced by a common name.
- ❖ Arrays can be one-dimensional, two-dimensional or multidimensional.
- ❖ An array is declared by specifying its base type, name, and size.
- ❖ The data type of array elements is known as the base type of the array.
- ❖ In Java, array indices start from 0 and are upto size 1.
- ❖ Array elements are stored in the contiguous memory locations.
- ❖ The total bytes required to store a one-dimensional array are determined by the formula :

$$\text{size of type} * \text{size of array}$$
- ❖ Two-dimensional arrays are stored in memory in a row-column matrix.
- ❖ The total number of bytes required to store a two-dimensional array are calculated as number of rows (first index) \times number of columns (second index) \times size of base type
- ❖ Arrays can be initialized at the time of declaration by providing the value list at the same time.
- ❖ In linear search, each element of the array is compared with the given item to be searched for, one by one.
- ❖ Binary search searches for the given item in a sorted array. The search segment reduces to half at every successive stage.
- ❖ Sorting of an array means arranging the array elements in a specified order.
- ❖ In selection sort, the smallest (or largest depending upon the desired order) key from the remaining unsorted array is searched for and put in the sorted array. This process repeats until the entire array is sorted.
- ❖ In bubble sort, the adjoining values are compared and exchanged if they are not in proper order. This process is repeated until the entire array is sorted.

Solved Problems

1. Which of the following array declarations are invalid ? State reasons for invalidity :

- (i) float a [+ 40]; (ii) int num [0-10]; (iii) double [50]
 (iv) amount [20] of float (v) int score [20] (vi) char name [30];
 (vii) float values [10];

Solution.

(i) Invalid. The array size must be unsigned integer constant. The correct form is :

float a [40];

(ii) Invalid. The size of array cannot be defined as 0 – 10. The correct form is :

int num[11];

(iii) Invalid. The name of array is missing. The correct form is :

double array_name [50];

(iv) Invalid. The base type cannot appear after array name and size. The correct form is :

float amount [20];

(v) Invalid. There should not be any blank space in between an array name and the first square bracket. The correct form is :

int score [20];

(vi) Valid.

(vii) Valid.

2. Suppose a 10-element array A contains the values a_0, a_1, \dots, a_9 . Find the values in A after each loop:

Solution.

(a) In the first iteration of the loop

$A[1] = A[0]$ is executed which sets $A[1] = a_1$, the value of $A[0]$.

In the next iteration, $A[2] = A[1]$ is executed which makes $A[2] = A[1]$ i.e., a_1 which is the current value of $A[1]$.

In the next iteration, $A[3] = A[2]$ which makes $A[3] = a$, which is the current value of $A[2]$.

The same process repeats for all elements.

Thus, after the loop shown in (a), every element of A will have the value a_1 , the original value of A [0].

(b) First iteration executes $A[9] = A[8]$ and set $A[9] = 2$, the value of $A[8]$.

The second iteration executes $A[9] = A[8]$ and sets $A[8]$ to a_9 , the value of $A[8]$.

The process repeats similarly for other elements and at the end of the loop the elements have values $a_1, a_1, a_2, a_3, \dots, a_9$.

3. Write a Java program to find the largest and smallest elements in a vector.

Solution.

```
import java.io.DataInputStream ;\n\nclass Array1      {\n    public static void main(String args[ ])  {\n        DataInputStream in = new DataInputStream(System.in) ;\n        final int size = 25 ;\n        int i, n = 0, large, small ;\n        int v[ ] = new int[size] ;\n        float avg = 0, total = 0 ;\n        try {\n            System.out.println("Enter how many elements(max 25) : ") ;\n            n = Integer.parseInt(in.readLine()) ;\n            System.out.println("Enter elements of vector - > ") ;\n            for(i = 0; i < n; i++)\n                v[i] = Integer.parseInt(in.readLine()) ;\n        }\n        catch (Exception e) { }\n        large = v[0] ;\n        small = v[0] ;\n        for(i = 0; i < n; i++) {\n            if(v[i] > large)\n                large = v[i];\n            else if(v[i] < small)\n                small = v[i];\n        }\n        System.out.println("Largest element : " + large) ;\n        System.out.println("Smallest element : " + small) ;\n    }\n}
```

4. Program to delete duplicate elements from a vector.
- Solution.**

```

import java.io.DataInputStream ;
class Array2 {
    public static void main(String args[ ]) {
        DataInputStream in = new DataInputStream(System.in) ;
        final int size = 25 ;
        int i, j, n = 0, k, ans = 0 ;
        int v[ ] = new int[size] ;
        try {
            System.out.println("Enter how many elements(max 25) : ") ;
            n = Integer.parseInt(in.readLine()) ;
            System.out.println("Enter elements of vector -> ") ;
            for(i = 0; i < n; i++)
                v[i] = Integer.parseInt(in.readLine()) ;
        }
        catch (Exception e) { }
        System.out.println("Original vector -> ") ;
        for(i = 0; i < n; i++)
            System.out.print(v[i] + "\t") ;
        for(i = 0; i < n-1; i++)
            for(j = i + 1; j < n; j++) {
                if(v[i] == v[j]) {
                    n = n - 1 ;
                    for(k = j; k < n; k++)
                        v[k] = v[k + 1] ;
                    ans = 1 ;
                    j = j - 1 ;
                }
            }
        if(ans == 0)
            System.out.println("\nVector is without duplicates") ;
        else {
            System.out.println("\nVector after deleting duplicates ->") ;
            for(i = 0; i < n; i++)
                System.out.print(v[i] + "\t") ;
        }
    }
}

```

5. The following numbers : (89, 20, 31, 56, 20, 64, 48) are required to be sorted using selection sort showing how the list would appear at the end of each pass.

Solution.

Step I.	20, 89, 31, 56, 20, 64, 48
Step II.	20, 20, 31, 56, 89, 64, 48
Step III.	20, 20, 31, 56, 89, 64, 48
Step IV.	20, 20, 31, 48, 89, 64, 56
Step V.	20, 20, 31, 48, 56, 64, 89
Step VI.	20, 20, 31, 48, 56, 64, 89
Step VII.	20, 20, 31, 48, 56, 64, 89

6. Let $A(n \times n)$ be a two-dimensional array. Write an algorithm to find the sum of all the elements which lie on either diagonal. For example, for the matrix shown below, your algorithm should output $68 = (1 + 6 + 11 + 16 + 4 + 7 + 10 + 13)$:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Solution.

```

1. Let sum = 0
   /* Add elements of 1st diagonal */
2. For i = 1 to n do           // for Java, 0 to n - 1 do
{
3.     Sum = Sum + A[i, i]
}
   /* Add elements of 2nd diagonal */
4. For i = 1 to m do           // for Java, 0 to m - 1 do
{
5.     Sum = Sum + A[i, n - i + 1]
}
/* Print the sum */

6. Print 'The sum of diagonal elements is', sum.

```

7. Given an array : 89, 20, 21, 56. Sort this array in ascending order using Bubble Sort.

Solution. Given array is 89, 20, 31, 56, 20

Bubble Sort (**Bold** elements depict that they are to be compared in the next pass.)

I	20, 89, 31, 56, 20
II	20, 31, 89, 56, 20
III	20, 31, 56, 89, 20
IV	20, 31, 56, 20, 89
V	20, 31, 56, 20, 89
VI	20, 31, 56, 20, 89
VII	20, 31, 20, 56, 89
VIII	20, 31, 20, 56, 89
IX	20, 31, 20, 56, 89
X	20, 20, 31, 56, 89

8. State condition(s) under which binary search is applicable.

Solution. For binary search

(i) the list must be sorted, (ii) lower bound and upper bound and the sort order of $\{x_1, x_2, \dots, x_n\}$

9. Comment on the efficiency of linear search and binary search in relation to the number of elements in the list being searched.

Solution. The linear search compares the *Search Item* with each element of the array, one by one. If the *search Item* happens to be in the beginning of the array, the comparisons are low, however, if the element to be searched for is one of the last elements of the array, this search technique proves the worst as so many comparisons take place. The binary search on the other hand, tries to locate the *search Item* in minimum possible comparisons, provided the array is sorted. This technique proves efficient in nearly all of the cases.

Glossary

Array Collection of variables of the same type that are referenced by a common name.

Base Type Data type of array elements.

Index Also called subscript. The number designating its position in array's ordering.

One-dimensional Array A list of finite number of homogeneous data elements.

Two-dimensional Array An array in which each element is itself an array.

Unsized Array The array in which first dimension is missing.

Sorting Arranging elements of a data structure in some order (ascending or descending).

Assignments

TYPE A : VERY SHORT/SHORT ANSWER QUESTIONS

1. What do you understand by an array ? What is the significance of arrays ?
2. What is meant by index of an element ? How are indices numbered in JAVA ?
3. Determine the total bytes required to store $B[17]$, a char array.
4. What is the data type of elements of an array called ?
5. How is one-dimensional array represented in memory ?
6. Determine the number of bytes required to store an int array namely $A[23]$.
7. An array element is accessed using

(a) a first-in-first-out approach	(b) the dot operator
(c) an element name	(d) an index number
8. Write a statement that defines a one-dimensional array called **amount** of type **double** that holds 10 elements.
9. The elements of a twenty-element array are numbered from _____ to _____ .
10. Element **amount[9]** is which element of the array ?

(a) the eighth	(b) the ninth	(c) the tenth	(d) impossible to tell.
----------------	---------------	---------------	-------------------------
11. Declare following arrays

(i) figures of 30 char	(ii) check of 100 short
(iii) balance of 26 float	(iv) budget of 58 double
12. Declare an array of 5 ints and initialize it to the first five even numbers.
13. Write a statement that displays the value of the second element in the long array **balance**.
14. For a multidimensional short array $X[5][24]$, find the number of bytes required.
15. For a multidimensional array $B[9][15]$ find the total number of elements in B .
16. How are the 2-D arrays stored in the memory ?
17. What are the preconditions for Binary Search to be performed on a single dimensional array ?

Or

State condition(s) when binary search is applicable.

18. State True or False :
 - (i) A binary search of an ordered set of elements in an array is always faster than a sequential search of the elements. True or False ?

- (ii) A binary search of elements in an array requires that the elements be sorted from smallest to largest. True or False ?
- (iii) The performance of linear search remains unaffected if an unordered array is sorted in ascending order. True or False ?
- (iv) Arrays do not prove to be useful where quite many elements of the same data types need to be stored and processed. True or False ?
- (v) For sorting of an element in an array, the elements are always shifted to make room for the new entrant. True or False ?
19. Name some commonly used sorting techniques.
20. Show the contents of the array

12	7	2	43	11	19	1	3	35	6
0	1	2	3	4	5	6	7	8	9

after the second iteration of Selection Sort.

21. Show the contents of the array

35	6	8	11	15	9	7	22	41	65
0	1	2	3	4	5	6	7	8	9

after the second iteration of Bubble Sort.

22. What is the difference between linear search and binary search ?

23. Given the following array :

27	23	3	18	24	1	12	60	47	5
0	1	2	3	4	5	6	7	8	9

Which sorting algorithm would produce the following result after three iterations ?

1	3	5	18	24	27	12	60	47	23
0	1	2	3	4	5	6	7	8	9

24. Given the following array :

13	19	6	2	35	28	51	16	65	4
0	1	2	3	4	5	6	7	8	9

Which sorting algorithm would produce the following result after three iterations ?

13	6	2	19	35	28	51	16	65	4
0	1	2	3	4	5	6	7	8	9

ARRAYS

TYPE B : LONG ANSWER QUESTIONS

- What is an array? What is the need for arrays?
- What are different types of arrays? Give examples of each array type.
- Write a note on how single-dimension arrays are represented in memory.
- How does the amount of storage (in bytes) depend upon type and size of an array? Explain with the help of an example.
- What do you understand by two-dimensional arrays? State some situations that can be easily represented by two-dimensional arrays.
- How are two-dimensional arrays represented in memory?
- Suppose A, B, C are arrays of integers of sizes m, n, m + n respectively. Give a program to produce a third array C, containing all the data of array A and B.
- Write a short program that doubles every element of an array A [4] [4].
- Let A ($n \times n$) that are not diagonal array. Write a program to find the sum of all the elements which lie on either diagonal. For example, for the matrix shown below, your program should output

$$68 = (1 + 6 + 11 + 16 + 4 + 7 + 10 + 13) :$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

- Write a function that checks whether or not two arrays (of characters) are identical, that is, whether they have same characters and all characters in corresponding positions are equal.
- Differentiate between one-dimensional and two-dimensional arrays.
- Explain :
 - Linear search method
 - Binary search method

Which of the two is more efficient for sorted data?
- Write a program Upper-half which takes a two dimensional array A, with size N rows and N columns as argument and point the upper half of the array.

Eg : If A is

2	3	1	5	0
7	1	5	3	1
2	5	7	8	1
0	1	5	0	1
3	4	9	1	5

The output will be

2	3	1	5	0
1	5	3	1	
7	8	1		
0	1			

- Write a program to search for an ITEM linearly in array X [10].
- Write a program to search for an ITEM using binary search in array X [10].
- The following array of integers is to be arranged in ascending order using the bubble sort technique :

26 21 20 23 29 17

Give the contents of the array at the end of each iteration. Do not write the algorithm.

- Write a program to search for a given ITEM in a given array X[n] using linear search technique. If the ITEM is found, move it at the top of the array. If the ITEM is not found, insert it at the end of the array.

18. Write a program to search for 66 and 71 in the following array :

3	4	7	11	18	29	45	71	87	89	93	96	99
0	1	2	3	4	5	6	7	8	9	10	11	12

Make use of binary search technique. Also give the intermediate results while executing this algorithm.

19. Given the following array :

13	7	6	21	35	2	28	64	45	3	5	1
0	1	2	3	4	5	6	7	8	9	10	11

Write a program to sort the above array using exchange selection sort. Give the array status after every iteration.

20. For the same array mentioned above in question 6, write a program to sort the above array using bubble sort technique. Give the array-status after every iteration.
21. What are the advantages and disadvantages of an array ?

CHAPTER 12

Operations on Files

12.1 Introduction

By now, you very well know that **System.in** and **System.out** objects are used for obtaining input from keyboard and for sending output to screen respectively. As you know that **System.in** and **System.out** are special stream objects which read data from the keyboard and write data to screen.

This chapter is going to take you on a farther journey in the *Java Streamsland*. You will be learning to use stream objects for reading from files and for writing onto files. In Java, if we wish to read and write to a file then we need to *replace* **in** and **out** objects with appropriate stream objects other than **in** and **out** objects.

12.2 Files

Till now whatever data you read and processed, resided in memory as long as you were executing your program. The moment your program got over or you switched off your computer, this data was removed from memory and you no longer could use or access this data. However, if you want to store this data for later use, you need to store them in files. Any data to be stored permanently, should be stored in files.

In This Chapter

-
- 12.1 Introduction
 - 12.2 Files
 - 12.3 Java Streams
 - 12.4 Operations on Files
 - 12.5 String Tokenizer
 - 12.6 Stream Tokenizer
 - 12.7 Obtaining Input using Scanner Class
 - 12.8 Printing in Java

There are at least three good reasons for structuring a collection of data as a file. These are as follows :

1. Any data or instructions which are to be processed upon, are to be presented in the main memory. After the processing ends, the operating system reallocates the space occupied by processed data/instructions to another program and the data / instructions of the first program are gone. By contrast, if these data/instructions are stored in a file on secondary storage, they provide permanent data storage. Even after the data/instructions are lost in the main memory, these are available on the secondary storage for later use.
2. Files can also store a data collection which is too large to fit in the available main memory.
3. Also, if a program requires only a small portion of data, then rest of the data reside on files securing data for later use. Therefore, a file is an important factor of information management of a computer system.

Let us now learn how we can read and write to files from within Java.

12.3 Java Streams

The stream classes that you use for input and output purposes, are contained in `java.io` package. Recall that there are two types of stream classes in Java :

- (i) *Byte stream classes used for Byte oriented IO.*
- (ii) *Character stream classes used for Character oriented IO.*

The **Byte stream classes** can be categorized into :

- ◆ **Input Stream Classes**
- ◆ **Output Stream Classes**

Each of the above two class types can further be categorized into *Memory*, *File* and *Pipe* type of streams.

The **Character stream classes** can be categorized into :

- ◆ **Reader classes**
- ◆ **Writer classes**

Each of the above two classes can further be categorized into *Memory*, *File* and *Pipe* type of streams. Figure 12.1 shows this categorization of Java streams.

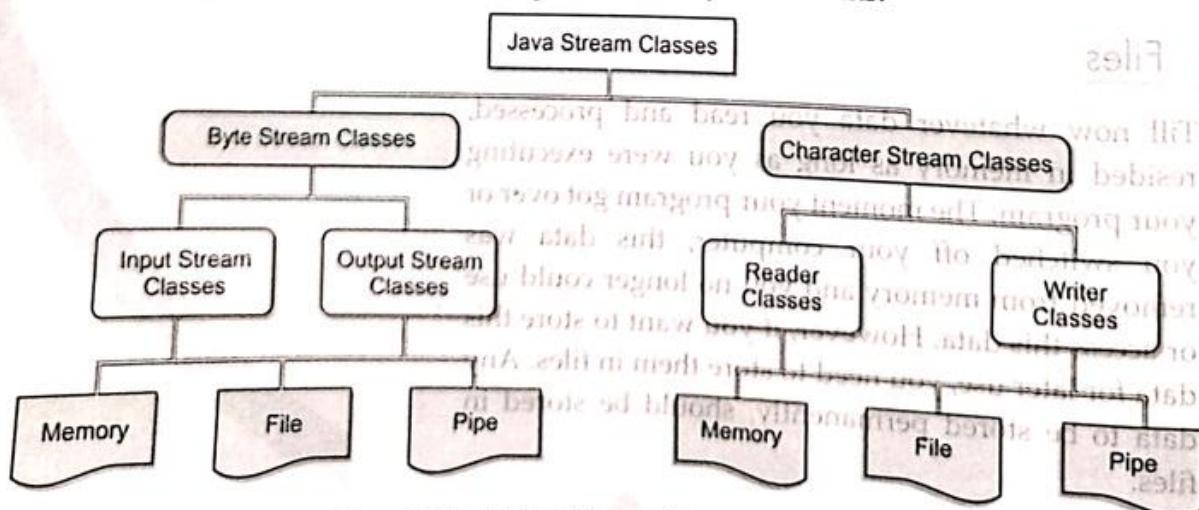


Figure 12.1 Various Stream Class types in Java.

In order to work with files, you need to create a file stream type objects and then link them to desired files; just as your standard input stream is linked to keyboard and your standard output stream is connected to a display device e.g., monitor. (Fig. 12.2)

You have already learnt to work with standard I/O streams in chapter 8. In this chapter, we shall learn how to perform file based I/O using streams.



Figure 12.2 Flow of data in standard I/O streams and file streams.

12.4 Operations on Files

With files, you can perform various operations on them e.g.,

- ◆ Obtaining input from a file (*File Input*)
- ◆ Writing output to a file (*File Output*)

These types of file operations can be performed on both types of files, i.e.,

- (i) **On Character files** (on text files) i.e., the files that store characters as per a specific character encoding scheme (Unicode in Java).
- (ii) **On Binary files** i.e., the files that store data in machine readable form.

Character Oriented IO is performed on *text files* whereas Byte Oriented IO is performed on *binary files*. Before we learn to perform IO on files, let us briefly talk about buffering—a concept commonly used with files.

12.4.1 Buffering

Accessing any input or output device usually requires some processing, independent of the amount of data being transferred. The *overhead* of accessing the device can become significant if very small amounts of data are being transferred very often. It is usually more efficient to transfer larger amounts less often. This can be achieved through the use of *buffering*. A *buffer* is a temporary storage used to hold data until enough has been collected that it is worth transferring. Buffering can be used for both input and output.

An **input buffer** is used for reading a large chunk of data from a stream. The buffer is then accessed as needed and, when emptied, another chunk of data is read from the stream into the buffer. Input buffering can also be used to return to earlier points in the stream (the mark and reset operations).

A **Buffer** is a temporary storage used to hold data until enough has been collected that it is worth transferring. Buffering can be used for both input and output.

An **output buffer** is used to store up data to be written to a stream. Once the buffer is full, the data is sent to the stream all at once and the buffer is emptied to receive more data.

Sometimes we wish to guarantee that some data has actually been written to the stream and that it is not waiting around in the output buffer. At such times, we execute a **flush** operation, which forces the buffer to be written to the stream as soon as possible.

The **BufferedInputStream** and **BufferedOutputStream** filters may be used to add buffering to any byte-oriented stream. The **BufferedReader** and **BufferedWriter** filters may be used to add buffering to any character-oriented stream.

Let us now learn to perform IO on text files.

12.4.2 Output to Text Files

Before you actually start using and manipulating files in your programs, let us discuss the steps you need to follow in order to use text files in your program. Following lines list the steps that need to be followed in the order as mentioned below. The steps to use text files in your Java program are :

The steps to use text files in your Java program are :

- Step 1.** Create a **FileWriter** stream type object as per following syntax :

```
FileWriter <stream-object-name> = new FileWriter ("<filenames>");
```

e.g.,

```
FileWriter fout = new FileWriter ("Names.txt");
```

A **FileWriter** object creates an output stream using the default encoding scheme.

- Step 2.** Link the **FileWriter object**, that you created just now, with a **BufferedWriter** object as per following syntax :

```
BufferedWriter <stream-object-name> = new BufferedWriter (<FileWriter object>);
```

e.g., BufferedWriter bout = new BufferedWriter(fout);



this is the **FileWriter object** you created, in step 1.

A **BufferedWriter** object creates a buffer for the stream created with **FileWriter**.

- Step 3.** Now link the **BufferedWriter Object** with a **PrintWriter** object as per following syntax :

```
PrintWriter <object-name> = new PrintWriter (<BufferedWriter Object>);
```

e.g.,

```
PrintWriter pout = new PrintWriter(bout);
```



BufferedWriter object

The **PrintWriter** object is responsible for writing characters on to the text file you linked with **FileWriter** object.

This way you have created a stream combination (Fig. 12.3) that will be used for combining filtering, buffering and content. In the above combination, the `PrintWriter` stream will provide the necessary filtering, the `BufferWriter` stream will provide the buffering and the `FileWriter` stream will provide the content.

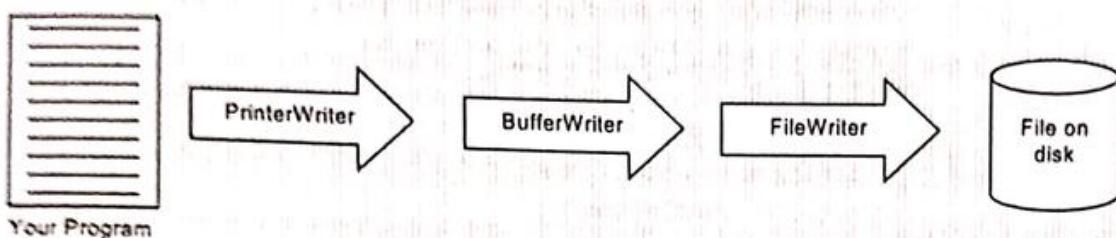


Figure 12.3 The stream combination of character oriented output.

Step 4. Now write any characters or text from your program with `PrintWriter` object using either `print` or `println` methods as :

```

<PrintWriter object>.print( ... ) ;
<PrintWriter object>.println( ... ) ;

e.g.,
pout.print("This is a") ;
pout.println("Test") ;
pout.println("This is a") ;
pout.println("Test") ;
  
```

The difference between `print` and `println` remains the same.

Step 5. Once you are through with everything, just close the stream chain by using `close()` with all stream objects :

```

<Stream object>.close( )

e.g.,
pout.close() ;
bout.close() ;
fout.close() ;
  
```

Note Make sure to import `java.io` package in all input-output programs.

Let us now put everything we have learnt so far, to test. Let us read names of 5 students and store them on file namely "names.txt".

Program 12.1

Read names of 5 students through keyword and store them in file names.txt.

```

import java.io.*;
public class IO {
    static String fileName = ("names.txt");
    static InputStreamReader isr = new InputStreamReader(System.in);
  
```

```

static BufferedReader stdin = new BufferedReader(isr);
public static void main(String[] args) {
    try {
        PrintWriter pw = new PrintWriter(fileName);
        BufferedWriter bw = new BufferedWriter(pw);
        PrintWriter outFile = new PrintWriter(bw);
        for(int i = 0; i < 5; i++) {
            System.out.print("Enter Name : ");
            String name = stdin.readLine();
            outFile.println(name);
        }
        outFile.close();
    } catch(IOException e) {
        System.err.println(e);
    }
}

```

The output produced is

Enter Name : Razia
 Enter Name : Sonu
 Enter Name : Prabhjyot
 Enter Name : Arnold
 Enter Name : Zubin

Appending to Text File

Now do one thing—Once again run the previous program: That is, once again provide the different input and then open *names.txt* file. You'll see that all your previous data is overwritten and the file stores only the data you typed in recent most execution. This is because of the default behaviour of character output streams.

Note By default, Output streams create with the specified file, if it does not exist. And if it already exists, the output streams overwrite the existing contents.

What if you want to retain the old contents and append the new contents?

Well, just create the *FileWriter* object as follows :

```
FileWriter <objectname> = new FileWriter(<filename>, true);
```

(<objectname>)

This means the new content will be appended to file and old data will be retained

This means the new data will be appended to file. If you skip it or write **false** here, the old data will be overwritten.

i.e., if you replace the line that creates *FileWriter object* with the following line :

```
FileWriter fout = new FileWriter("names.txt", false);
```

Your new data will get appended to the file. Make these changes and re-run the above program. See what happens.

To flush all the buffered data to the file i.e., to write to the file whatever data is there in the buffer, you can use :

```
<PrintWriter object>.flush();
```

e.g., *pout.flush()*;

will flush all buffered data to the linked file.

12.4.3 Input from Text Files

To read from a text file, you need to follow these steps : (Make sure the file you are planning to read from, exists on the disk, otherwise an exception will be thrown).

- Step 1.** Create a **FileReader** stream type object as per following syntax :
- ```
FileReader <stream-object-name> = new FileReader ("filename");
```
- e.g.,
- ```
FileReader fIn = new FileReader ("Names.txt");
```
- A **FileReader** object creates an input stream using the default **encoding scheme**.
- Step 2.** Link the **FileReader object**, that you created just now, with a **BufferedReader** object as per following syntax :
- ```
BufferedReader <stream-object-name> = new BufferedReader (<FileReader object>);
```

e.g.,

```
BufferedReader bin = new BufferedReader(fIn);
```

This is the **FileReader object** you created, in step 1.

A **BufferedReader object** creates a buffer for the stream created with **FileReader**.

- Step 3.** Now you can read text using **readLine()** from this **BufferedReader object** and store in a **string object** e.g.,

```
String txt = bin.readLine();
```

- Step 4.** Use the read text in the program as required.

- Step 5.** Once you are through, close the file as

```
<BufferedReader object>.close();
```

e.g.,

```
bin.close();
```

Let us now put everything we have learnt in the form of a complete program. But wait how would you make out that all the data has been read ?

When you use **readLine()** with a **BufferedReader**, if no more data is found, a **null** is returned, thus we shall check for this very thing i.e., as

```
while ((text = bin.readLine()) != null)
{
 : // process data here
}
```

But make sure not to include **bin.readLine()** in the loop-body, otherwise two lines will be read but one would be processed.

Let us now move on to the complete program.

### Program 12.2

Read data from text file "names.txt" and display it on monitor.

```
import java.io.*;
class FileIOExample2{
 public static void main(String[] args) throws IOException {
 FileReader file = new FileReader("names.txt");
 BufferedReader fileInput = new BufferedReader(file);
 String text;
 // Read file and output
 int i = 0;
 while ((text = fileInput.readLine()) != null) {
 i++;
 System.out.print("Name " + i + " : ");
 System.out.println(text);
 }
 fileInput.close(); // Close file
 }
}
```

The output produced is

|        |   |          |
|--------|---|----------|
| Name 1 | : | Razia    |
| Name 2 | : | Shyla    |
| Name 3 | : | Khan     |
| Name 4 | : | Gurpreet |
| Name 5 | : | Michael  |

#### 12.4.4 Output to Binary Files

After learning to work with text files, let us now learn to perform IO on binary files *i.e.*, to perform byte oriented IO. Steps to perform byte oriented IO are similar, only the stream classes to be used are different.

The steps to send output to a binary file are :

- Step 1.** Create a `FileOutputStream` type object and link it with the file name to which output is directed.  
The syntax for doing so is as follows :

e.g.,  
`FileOutputStream <objectname> = new FileOutputStream (<filename>);`

- Step 2.** Now connect the `FileOutputStream` object with a `DataOutputStream` as per following syntax:  
e.g.,  
`DataOutputStream <objectname>`  
`= new DataOutputStream(<FileOutputStream object>);`

`DataOutputStream dout = new DataOutputStream(fout);`

*this is the FileOutputStream object we created in step 1.*

- Step 3.** Now you can write data to file using either `write()` function or `flush()` function. The `write()` function writes data in byte form on the file and `flush()` writes all the buffered data on to the file.

**Note** Make sure that file you are reading from resides in the same folder as that of the program.

Various write( ) functions are :

|                    |                       |
|--------------------|-----------------------|
| writeByte(byte)    | writeShort(short)     |
| writeInt(int)      | writeLong(long)       |
| writeFloat(float)  | writeDouble(double)   |
| writeChar(char)    | writeBoolean(boolean) |
| writeBytes(string) | writeChars(string)    |
| writeUTF(string)   |                       |

To write Unicode Transformed Format strings

The datatype in brackets indicate the type of data the function can write.

- Step 4.** Once through, close all the stream objects using close( ) e.g.,  
 dout.close( );  
 fout.close( );

Let us now write a complete program to write some data on a binary file.

#### Program 12.3

Read rollno and marks of 5 students and write them on a file namely "stu.dat".

```
import java.io.* ;
public class BinaryOutput {
 static String fileName = "stu.dat" ;
 static InputStreamReader isr = new InputStreamReader(System.in) ;
 static BufferedReader stdin = new BufferedReader(isr) ;
 public static void main(String[] args) {
 try {
 int rno ; float marks ;
 FileOutputStream fw = new FileOutputStream(fileName) ;
 DataOutputStream dw = new DataOutputStream(fw) ;
 for(int i = 0 ; i < 5 ; i++)
 {
 System.out.print("Enter Rollno :") ;
 rno = Integer.parseInt(stdin.readLine()) ;
 System.out.print("Enter Marks :") ;
 marks = Float.parseFloat(stdin.readLine()) ;
 dw.writeInt(rno) ;
 dw.writeFloat(marks) ;
 }
 dw.close() ;
 fw.close() ;
 }
 catch (IOException e)
 {
 System.err.println(e) ;
 }
 }
}
```

The output produced is

|                |    |
|----------------|----|
| Enter Rollno : | 1  |
| Enter Marks :  | 98 |
| Enter Rollno : | 2  |
| Enter Marks :  | 87 |
| Enter Rollno : | 3  |
| Enter Marks :  | 96 |
| Enter Rollno : | 4  |
| Enter Marks :  | 74 |
| Enter Rollno : | 5  |
| Enter Marks :  | 69 |

Now open the folder where you have stored your program. See a file namely *stu.dat* has been created. Try opening this file with the help of Notepad. Can you read the contents? No? Well, this is the beauty(?) of binary files.

Recall that if the file does not exist, it will be created and if it already exists, by default, all old data gets overwritten. To append data on to a file, provide an addition boolean *true* argument while creating *FileOutputStream* object.

#### 12.4.5 Input from Binary Files

To read from an already existing binary file, following steps are to be followed :

**Step 1.** Create a *FileInputStream* object and connect it to an existing binary file as follows :

```
FileInputStream fin = new FileInputStream("Stu.dat");
```

*FileInputStream object name*

*File name*

**Step 2.** Connect this *FileInputStream* object with a *DataInputStream* object as follows :

```
DataInputStream din = new DataInputStream(fin);
```

*DataInputStream object*

*FileInputStream object that we*

created in Step 1

**Step 3.** Now read data from this *DataInputStream* object using any of the following *read()* functions :

|                      |                                         |
|----------------------|-----------------------------------------|
| <i>readByte()</i>    | returns byte                            |
| <i>readShort()</i>   | returns short                           |
| <i>readInt()</i>     | returns int (-2147483648 to 2147483647) |
| <i>readLong()</i>    | returns long                            |
| <i>readFloat()</i>   | returns float                           |
| <i>readDouble()</i>  | returns double                          |
| <i>readChar()</i>    | returns char                            |
| <i>readBoolean()</i> | returns boolean                         |
| <i>readLine()</i>    | returns string                          |
| <i>readUTF()</i>     | returns UTF string                      |

The input streams are automatically closed when end-of-file is reached. Upon reaching end-of-file *EOFException* is thrown and input file stream is automatically closed.

#### Checking for Binary File EOF

Checking for a binary file's end-of-file condition is a tricky task because reading (*read()*) does not return -1 or null as in the case of text files. It only throws *EOFException* and closes the input file stream.

Now to check the end of file, write a code somewhat like;

```

1: boolean eof = false ;
2: while(!eof) {
3: try {
4: BufferedReader br = new BufferedReader(new FileReader("C:\\" + "file.txt"));
5: String str = br.readLine();
6: if(str == null)
7: eof = true;
8: } catch (IOException e) {
9: System.out.println("Error reading file");
10: }
11: }

```

This functionality has been used in following program 12.4.

Let us now write a program to read the binary file that we created in program 12.4.

### Program 12.4

**Read data from a binary file stu.dat that stores rollno and marks of some students.**

```

import java.io.*;
public class BinaryInput {
 static String fileName = "Stu.dat";
 public static void main(String[] args) {
 boolean EOF = false;
 try {
 FileInputStream fr = new FileInputStream(fileName);
 DataInputStream dr = new DataInputStream(fr);
 while(!EOF)
 {
 try {
 int rno ; float marks ;
 rno = dr.readInt();
 System.out.println("Rollno :" + rno);
 marks = dr.readFloat();
 System.out.println("Marks :" + marks);
 } catch (EOFException el) {
 }
 System.out.println("end of file");
 EOF = true;
 }
 } catch (IOException e) {
 }
 }
}

```

```

 catch (IOException e)
 {
 System.err.println(e);
 }
 } // end of while
} // end of outer try block

catch (FileNotFoundException e)
{
 System.out.println("File not found");
}
} // end of main
} // end of class

```

The output produced is

|             |      |
|-------------|------|
| Rollno :    | 1    |
| Marks :     | 98.0 |
| Rollno :    | 2    |
| Marks :     | 87.0 |
| Rollno :    | 3    |
| Marks :     | 96.0 |
| Rollno :    | 4    |
| Marks :     | 74.0 |
| Rollno :    | 5    |
| Marks :     | 69.0 |
| end of file |      |

**Note** Make sure the file being read exists in the same folder as that of the program (reading this file) itself.

## 12.5 String Tokenizer

Usually we input values a value at a time following each value by a carriage return. In some cases it is nice to be able to input values as a sequence. We can do this by simply entering the sequence as a string. However, once Java has got the string, to do anything useful with it we must be able to isolate different values within the string. We could do this by painstakingly analysing the string character by character (it is after all a character array) and find the delimiters for each "word" (double quotes are used here as the word in question may in fact be a number or some sequence of special characters). Such words are referred to as *tokens*.

Some example code to achieve this is given below in program 12.5.

### Program 12.5

*Input a string and separate words in it. Process the string as a character array.*

```

import java.io.*;
import java.util.*;
class StringProcessing {
 static InputStreamReader input = new InputStreamReader(System.in);
 static BufferedReader keyboardInput = new BufferedReader(input);
 public static void main(String[] args) throws IOException {
 // Get a string
 System.out.println("Input a string");
 String data = keyboardInput.readLine();
 // Output number of characters in the line
 int numberCharacters = data.length();
 System.out.println("Number of characters =" + numberCharacters + "\n");
 // Output tokens
 for (int counter = 0 ; counter < numberCharacters ; counter++) {
 char character = data.charAt(counter);
 }
 }
}

```

```

 if(character == ' ')
 System.out.println();
 else
 System.out.print(character);
 }
 System.out.println("\n");
}
}

```

A better way of doing this is to use the  **StringTokenizer** class, which is found in the Java package *util*. This contains a number of useful instance methods that can be used to isolate tokens (Fig. 12.4).

To use these methods we must of course first create an instance of the class  **StringTokenizer** as shown below :

```
StringTokenizer data = new StringTokenizer(String);
```

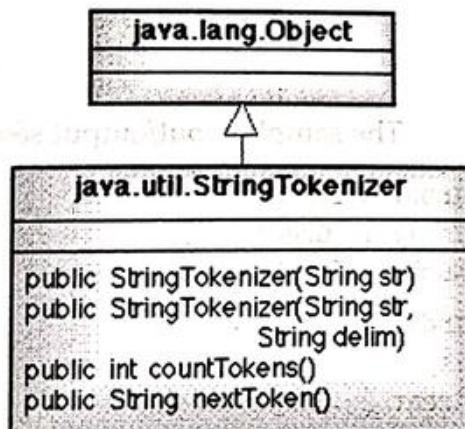
where the argument is a string of text. The  **StringTokenizer** class also provides two methods that are immediately useful for processing strings :

1.  **countTokens** method which returns the number of tokens that are delimited by any white space in a given string, thus we know how many tokens there are and therefore can use this number as a loop parameter with which to process the string.
2.  **nextToken** method which returns the next token in a string form the current token. When used for the first time the next token is the first token in the string. Thus this method can be used to pass along the string token by token.

You can use these methods with the  **StringTokenizer** object.

In the following program, program 12.6, we are using these methods to identify and output the tokens in a string provided by the user.

The output produced is  
 Input a string  
 A Sample String  
 Number of characters = 15



**Figure 12.4** Class diagram showing details of the  **StringTokenizer** class.

**Note**  **StringTokenizer** class is found in **java.util** package.

### Program 12.6

Program to illustrate usage of methods inside  **StringTokenizer** class.

```

import java.io.*;
import java.util.*;
class TokenizerExample {
 static InputStreamReader input = new InputStreamReader(System.in);
 static BufferedReader keyboardInput = new BufferedReader(input);
 public static void main(String[] args) throws IOException {
 int numberOfTokens = 0;

```

```

 System.out.println("Input a string");
 StringTokenizer data = new StringTokenizer(keyboardInput.readLine());
 print // Output number of tokens in the line
 numberoftokens = data.countTokens();
 System.out.println("Number of tokens = " + numberoftokens + "\n");
 // Output tokens
 for (int counter = 0; counter < numberoftokens; counter++) {
 System.out.println(data.nextToken());
 }
}

```

|             |
|-------------|
| JavaIO.java |
|             |

The sample input/output session of above code is as shown below :

|                    |
|--------------------|
| Input a string     |
| India is Great     |
| Number of tokens=3 |
| India              |
| is                 |
| Great              |

### 12.5.1 Processing a Number Sequence with StringTokenizer

In the above program, we assumed that words are delimited (*i.e.*, separated) by a space, which is the default delimiter. What if you need to process something which has a delimiter other than the space? Well, in that case just go through our next program (program 12.7). The java code presented program 12.7 processes a sequence of comma separated integers using the methods found in the StringTokenizer class. As you know, by default the delimiter is a white space character, in this case we have specified the nature of the delimiter, *i.e.*, a comma, as part of the constructor. Note that, whatever the delimiter is defined as, it is not considered to be a token in its own right. Note also that we are using the Integer wrapper class methods to convert the individual tokens from strings to integers.

#### Program 12.7

Program to use StringTokenizer class for a sequence delimited by comma.

```

import java.io.*;
import java.util.*;
class TokenizerExample2{
 static InputStreamReader input = new InputStreamReader(System.in);
 static BufferedReader keyboardInput = new BufferedReader(input);
 public static void main(String[] args) throws IOException {
 int numberoftokens = 0;
 int numberarray[];
 int total = 0;
 // Get a string
 System.out.print("Input a sequence of integers separated by commas(,') :");
 }
}

```

```

 StringTokenizer data = new StringTokenizer(keyboardInput.readLine(), ",") ;
 // Get number of tokens in line and initialise array
 numberOfTokens = data.countTokens() ; ; 0 = i thi
 System.out.println("Number of tokens = " + numberOfTokens + "\n") ;
 numberArray = new int[numberOfTokens] ;
 // Isolate tokens and maintain total
 for (int counter = 0 ; counter < numberOfTokens ; counter++) {
 numberArray[counter] = new Integer(data.nextToken()) .intValue() ;
 System.out.println(numberArray[counter]) ;
 total = total + numberArray[counter] ; ; () ??
 }
 // Now Output total
 System.out.println("----\n" + total +
 "(average = " + total/numberOfTokens + ")");
}

```

Program 12.8 String Tokenizer

The sample input/output session of above program is as shown below:

**Input a sequence of integers separated by commas( ', ' ) :2,4,3,45,32,42,99**

**Number of tokens =7**

2

4

3

45

32

42

99

227(average =32)

We are keeping our discussion limited to single delimiters.

### 12.5.2 String Tokenizing and File Handling

We can also use the *StringTokenizer* to process input from a file line by line as shown in program 12.8. Here we read a file called *names.txt* in the same manner as we did in program 12.2 and then use the tokenizer to identify and output the contents.

#### Program 12.8

Program to use *StringTokenizer* with a text file.

```

import java.io.*;
import java.util.*;
class FileAndStringTokenizer {
 public static void main(String[] args) throws IOException {
 FileReader file = new FileReader("names.txt");
 BufferedReader fileInput = new BufferedReader(file);
 }
}

```

```

 String text ;
 // Read file and output
 int i = 0 ;
 int numberOfTokens = 0 ;
 while((text = fileInput.readLine()) != null) {
 StringTokenizer dataLine = new StringTokenizer(text) ;
 numberOfTokens = numberOfTokens + dataLine.countTokens() ;
 }
 // Output result and close file
 System.out.println("Number of tokens =" + numberOfTokens) ;
 fileInput.close() ;
 }
}

```

The output produced is  
Number of tokens = 5

## 12.6 Stream Tokenizer

We have seen that when reading an input string supplied by a user we like to be able to analyse it token by token as demonstrated in programs 12.6 and 12.7. To isolate such tokens we used the StringTokenizer class. We can also use the string tokenizer to process input from a file line by line as also demonstrated in program 12.8. However, there are a number of problems with this approach :

1. We would like to be able to detect the end of the file (EOF) without having to know in advance either :

The number of lines in the input file, or

Relying on special symbols (such as a blank line) to indicate the EOF.

2. It would also be nice to be able to process a file token by token, rather than a line at a time, should this be desirable.

To address the above we can make use of the *StreamTokenizer* class. Let us see how.

A stream tokenizer takes an input stream and parses it into tokens, allowing the tokens to be read one at a time. A partial class diagram for the *StreamTokenizer* class is given in Fig. 12.5.

Let us now learn to use *StreamTokenizer* with the help of an example code given in program 12.9 below.

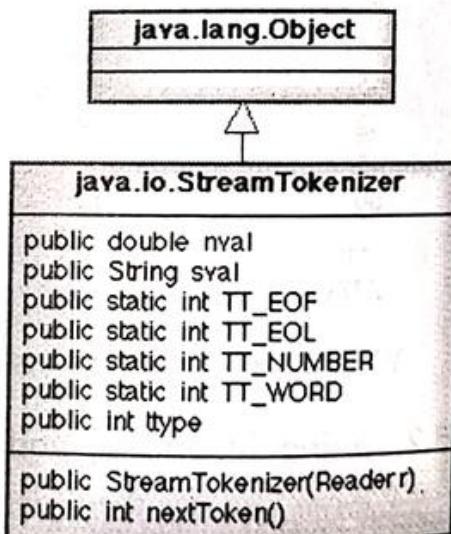


Figure 12.5 Class diagram showing continuation some of the Stream Tokenizer fields and methods.

### Program 12.9

Program to illustrate usage of *StreamTokenizer* class.

```

import java.io.*;
import java.util.*;
class TokenizerExample16_9 {
 public static void main(String[] args) throws IOException {

```

```

FileReader file = new FileReader("TokenizerExample16_9.java");
StreamTokenizer inputStream = new StreamTokenizer(file);
int tokenType = 0 ;
int numberOfTokens = -1 ;
// Process the file and output the number of tokens in the file
do {
 tokenType = inputStream.nextToken();
 numberOfTokens++;
}
while (tokenType != StreamTokenizer.TT_EOF) ;
// Output result and close file
System.out.println("Number of tokens =" + numberOfTokens) ;
}
}

```

The code contains a constructor which requires an argument of the type *FileReader*. Having created an instance of the class *streamTokenizer* we can use the *nextToken* method to read tokens from the input stream. Note also that we do not need to know how big the file is in advance in this case, we simply test the current token's type against the class integer constant *TT\_EOF* (this has a value of -1). There are four possible predefined types of tokens : *TT\_EOF*, *TT\_EOL*, *TT\_Number* and *Word*.

Notice that the input file used by the program 12.9 is its own source code file *TokenizerExample16\_9.java*. And the output produced is as shown below :

Number of tokens =99

### Identifying Type of Tokens and Associated Values

We can modify the code of program 12.9 in a way so that it identifies the type of tokens and stores it appropriately for desired processing. Have a look at program 12.10 that accomplishes this very task. In program 12.10, we have added some code that identifies tokens and outputs the associated value. Firstly go through program 12.10 carefully.

### Program 12.10

Using *StreamTokenizer* to separate tokens of various types.

```

import java.io.*;
import java.util.*;
class TokenizerExample16_10 {
 public static void main(String[] args) throws IOException {
 FileReader file = new FileReader("sample.txt");
 StreamTokenizer inputStream = new StreamTokenizer(file);
 int tokenType = 0 ;
 int numberOfTokens = -1 ;
 // Process the file and output the number of tokens in the file
 do {
 tokenType = inputStream.nextToken();
 outputTtype(tokenType, inputStream);
 }
 }
}

```

```

 numberoftokens++ ;
 } while (tokenType != StreamTokenizer.TT_EOF) ;
 // Output result and close file
 System.out.println("Number of tokens =" + numberoftokens);
}

/* OUTPUT TTYPE : Method to output the ttype of a stream token and its value. */
private static void outputTtype(int ttype, StreamTokenizer inStream) {
 switch (ttype) {
 case StreamTokenizer.TT_EOF : System.out.println("TT_EOF");
 break;
 case StreamTokenizer.TT_EOL : System.out.println("TT_EOL");
 break;
 case StreamTokenizer.TT_NUMBER : System.out.println
 ("TT_NUMBER : nval =" + inStream.nval);
 break;
 case StreamTokenizer.TT_WORD : System.out.println ("TT_WORD : sval =" + inStream.sval);
 break;
 default : System.out.println("Unknown : nval =" + inStream.nval + "sval ="
 + inStream.sval);
 break;
 } // end switch
} // end outputTtype() function
} // end class

```

In program 12.10, if a token is of type *TT\_NUMBER* the value has been stored in the instance variable *nval*, and if it is of type *TT\_Word* in the instance variable *sval*. The result from running the code is presented below :

```

TT_WORD : sval =int
TT_WORD : sval =i
Unknown : nval =0.0sval =null
TT_NUMBER : nval =0.0
Unknown : nval =0.0sval =null
TT_WORD : sval =while
Unknown : nval =0.0sval =null
Unknown : nval =0.0sval =null
TT_WORD : sval =text
Unknown : nval =0.0sval =null
TT_WORD : sval =fileInput.readLine
Unknown : nval =0.0sval =null
TT_WORD : sval =null
Unknown : nval =0.0sval =null
TT_EOF
Number of tokens =21

```

Comparison of the above output with respect to the input file (sample.txt) shown above indicates that :

- ❖ All comments are ignored by the stream tokenizer.
- ❖ Characters such as \*, ;, (,), {}, [, ], = and + are tokens in their own right but do not have a specific `ttype` value (their `ttype` value is their ASCII value).
- ❖ Strings are output as strings without the double quotes and with carriage returns (`\n`) if encountered. The type value for a string is 34, which is the ASCII value for a double quote.
- ❖ End of line carriage returns are not considered to be tokens (we have no tokens of type `TT_EOL`).

Thus, using the stream tokenizer we could produce some code to "parse" a java source file and identify (say) the class and method names within it.

## 12.7 Obtaining Input using Scanner Class

The `Scanner` class, introduced with J2SE 5.0, is a very useful new class that can parse text for primitive types and substrings using regular expressions. It can obtain the text from sources such as a `String` object, an `InputStream`, a `File`, and any class that implements the `Readable` interface.

The `Scanner` splits input into substrings, or tokens, separated by delimiters, which by default consist of any white space. The tokens can then be obtained as strings or as primitive types if that is what they represent.

For example, the following code snippet shows how to read an integer from the keyboard

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

### Reading and Type Checking Tokens

For each of the primitive types there is a corresponding `nextXxx()` method that returns a value of that type. If the string cannot be interpreted as that type, then an `InputMismatchException` is thrown. A `Scanner` object will divide a line of input into tokens (fields) using white space to delimit the tokens. For example, given the line :

`bina 10 true`

a `Scanner` will create the tokens "`bina`", "`10`", and "`true`". You can retrieve these tokens and convert them to the appropriate types using the following set of methods :

|                                                                        |                                                                  |
|------------------------------------------------------------------------|------------------------------------------------------------------|
| ❖ <code>String next( ) :</code>                                        | Returns the next token as a string                               |
| ❖ <code>int nextInt( ) :</code>                                        | Returns the next token as an integer                             |
| ❖ <code>int nextBoolean( ) :</code>                                    | Returns the next token as a boolean                              |
| ❖ <code>float nextFloat,</code><br><code>double nextDouble( ) :</code> | Returns the next token in the appropriate floating point format. |

If the input has been exhausted or if the token cannot be converted to the desired type, then the above methods will throw exceptions. To avoid having exceptions thrown, you can check

whether or not a token exists and whether or not it conforms to the desired type using the following set of methods :

|                              |                                                                                                                          |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| ▢ boolean hasNext( ):        | Returns true if the input has another token                                                                              |
| ▢ boolean hasNextInt( ):     | Returns true if the input has another token and<br>and that token can be interpreted as an integer.                      |
| ▢ boolean hasNextBoolean( ): | Returns true if the input has another token and<br>that token can be interpreted as a boolean.                           |
| ▢ boolean hasNextFloat( ):   | Returns true if the input has another token and that token<br>can be interpreted as a floating point number.             |
| ▢ boolean hasNextDouble( ):  | Returns true if the input has another token and that token<br>can be interpreted as a double word floating point number. |

Following program demonstrates how to read int, float, and double values from the keyboard using a Scanner class.

### Program 12.11

Input some numbers with the help of Scanner class.

```

import java.io.*;
import java.util.*;
public class ScanConsoleApp
{
 public static void main (String arg[]) {
 // Create a scanner to read from keyboard
 Scanner scanner = new Scanner (System.in); ↗

 try {
 System.out.printf ("Input int (e.g. %4d): ",3501);
 int int_val = scanner.nextInt ();
 System.out.println (" You entered " + int_val +"\n");

 System.out.printf ("Input float (e.g. %5.2f): ", 2.43);
 float float_val = scanner.nextFloat ();
 System.out.println (" You entered " + float_val +"\n");

 System.out.printf ("Input double (e.g. %6.3e): ",4.943e15);
 double double_val = scanner.nextDouble ();
 System.out.println (" You entered " + double_val +"\n");
 }

 catch (InputMismatchException e)
 {
 System.out.println ("Mismatch exception:" + e);
 }
 } // main
} // class ScanConsoleApp

```

A session with this program goes as follows :

```
Input int (e.g. 3501): 23431
You entered 23431
Input float (e.g. 2.43): 1.2343
You entered 1.2343
Input double (e.g. 4.943e+15): -2.34e4
You entered -23400.0
```

There are a number of other useful methods in the **Scanner** class such as **skip()** to jump over input, **useDelimiter()** defines a delimiter in place of the default white space, and **findInLine()** to search for substrings.

### Discarding a Scanner

When you have finished using a Scanner, such as when you have exhausted a line of input or reached the end of a file, you should close the Scanner using the **close** method. If you do not close the Scanner then Java will not garbage collect the Scanner object and you will have a memory leak in your program.

Thus to close the scanner, use the following method :

◆ **void close( )** : closes the Scanner and allows Java to reclaim the Scanner's memory

Following program uses Scanner class to obtain input inside a loop.

### Program 12.12

Program to obtain numbers and display their sum.

```
import java.util.*;
public class ScannerLoop {
 public static void main(String[] args) {
 //... Initialization
 double n; // Holds the next input number.
 double sum = 0; // Sum of all input numbers.
 Scanner in = new Scanner(System.in);

 //... Prompt and read input in a loop.
 System.out.println("Enter number. Non-number will stop input.");
 while (in.hasNextDouble()) {
 n = in.nextDouble();
 sum = sum + n ;
 }
 in.close();
 //... Display output
 System.out.println("The total is " + sum);
 }
}
```



## Using Scanner for Input from a File

We earlier discussed using the Scanner class to obtain input from the console. The Scanner class can read from a file just as easily as it can from the console. To understand this, just go through the following example program.

The program uses Scanner to scan past the text at the beginning of the file and then reads each of the primitive type values. There are many options with the pattern matching capabilities of Scanner to jump past the initial text. We choose a simple technique of looking for the first primitive type value, which is a boolean type. So we loop over calls to `hasNextBoolean()` until we find a boolean value.

This method, and the similar ones for other primitive types, look ahead at the next token and return true or false depending on whether the token is of the type indicated. It does not jump past the token. So, if the next token is not a boolean, we use the `next()` method to skip this token and examine the next one. When we find the boolean we break out of the loop.

### Program 12.13

Program to obtain input from a file using Scanner class.

```
import java.io.*;
import java.util.*;
public class ScanFileApp {
 public static void main (String arg[]) {
 File file = null;
 if (arg.length > 0) file = new File (arg[0]) ; // Get the file from the argument line.
 if (file == null) { // or use the default
 file = new File ("textOutput.txt") ;
 }
 Scanner scanner = null ;
 try {
 scanner = new Scanner (file) ; // Create a scanner to read the file
 } catch (FileNotFoundException e) {
 System.out.println ("File not found!") ;
 System.exit (0) ; // Stop program if no file found
 }
 try {
 while (true) { // Skip tokens until the boolean is found.
 if (scanner.hasNextBoolean ()) break ;
 scanner.next () ;
 }
 System.out.printf ("Skip strings at start of %s %n", file.toString ()) ;
 System.out.printf ("and then read the primitive type values: %n%n") ;
 // Read and print the boolean
 System.out.printf (" boolean = %9b %n", scanner.nextBoolean ()) ;
 // and then the set of numbers
 System.out.printf (" int = %9d %n" , scanner.nextInt ());
 System.out.printf (" int = %9d %n" , scanner.nextInt ());
 }
 }
}
```

```

 System.out.printf (" int = %9d \n" , scanner.nextInt ());
 System.out.printf (" long = %9d \n" , scanner.nextLong ());
 System.out.printf (" float = %9.1f \n" , scanner.nextFloat ());
 System.out.printf (" double = %9.2e \n" , scanner.nextFloat ());
 }
 catch (InputMismatchException e) {
 System.out.println ("Mismatch exception:" + e);
 }
} // main
} // class ScanFileApp

```

The output of the program goes as :

```

Skip strings at start of textOutput.txt
and then read the primitive type values :
boolean = false
int = 114
int = 1211
int = 1234567
long = 987654321
float = 983.6
double = - 4.30e - 15

```

## 12.8 Printing in Java

Apart from providing various classes for obtaining input, Java also provides classes for printing on printer through Java 2D printing API. The Java 2D printing API is not limited to printing graphics. It enables you to print the content of an application's user interface as well. Content can be printed by sending raw data to the printer under formatting control of Java 2D printing API. The main classes and interfaces involved in printing are represented in the `java.awt.print` package<sup>1</sup>.

### 12.8.1 Java Printing API

Printing involves deciding what to print and then readying printer for it. In other words, we can say that printing involves creation of print jobs. A *print job* is a unit of work to be run on a printer and can consist of printing one or more files. The system assigns a unique job number to each job as it is received.

In the Java Printing System or the Java Printing API, the program has a contract with the printing subsystem to supply a given page at a given time. The printing subsystem may request that your application render a page more than once, or render pages out of sequence. This model provides several advantages :

1. By sending strips of the page instead of the whole page to the printer, it allows the application to print complex documents that would require more printer memory than is available. The application does not have to know how to print each strip; it
- 
1. The package `javax.print` also contains classes and interface that allow you to get access to the printing services. But we won't be discussing this package here as it is way beyond the scope of this book.

only needs to know how to render a given page. The API will take care of the rest. In this case, the printing subsystem might request that a page be rendered several times depending on the number of strips required to completely print the page.

- If the paper tray on a particular printer outputs the pages in reverse order, then your application might be asked to print the document in reverse order, so it will appear in the right order in the output tray.

### Rendering models

There are two printing models in Java: *Printable jobs* and *Pageable jobs*.

#### 1. Printables

Printable jobs are the simpler of the two printing models. This model only uses one *PagePainter* for the entire document. Pages are rendered in sequence, starting with page zero. When the last page prints, your *PagePainter* must return the NO SUCH PAGE value. The print subsystem will always request that the application render the pages in sequence. As an example, if your application is asked to render pages five through seven, the print subsystem will ask for all pages up to the seventh page, but will only print pages five, six, and seven. If your application displays a print dialog box, the total number of pages to be printed will not be displayed since it's impossible to know in advance the number of pages in the document using this model.

#### 2. Pageables

Pageable jobs offer more flexibility than Printable jobs, as each page in a Pageable job can feature a different layout. Pageable jobs are most often used with Books, a collection of pages that can have different formats. A Pageable job has the following characteristics :

- Each page can have its own painter. For example, you could have a painter implemented to print the cover page, another painter to print the table of contents, and a third to print the entire document.
- You can set a different page format for each page in the book. In a Pageable job, you can mix portrait and landscape pages.
- The print subsystem might ask your application to print pages out of sequence, and some pages may be skipped if necessary. Again, you don't have to worry about this as long as you can supply any page in your document on demand.
- The Pageable job doesn't need to know how many pages are in the document.

All of the classes required to print are located in the `java.awt.print` package, which is composed of three interfaces and four classes. The following tables define the classes and interfaces of the print package.

### Classes and Interfaces inside `java.awt.print` package

| Name       | Type  | Description                                                                                                                                                   |
|------------|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Paper      | Class | This class defines the page's physical characteristics.                                                                                                       |
| PageFormat | Class | PageFormat defines the page's size and orientation. It also defines which Paper to use when rendering a page.                                                 |
| PrinterJob | Class | This class manages the print job. Its responsibilities include creating a print job, displaying a print dialog box when necessary, and printing the document. |

| Name            | Type      | Description                                                                                                                                                                                                                                 |
|-----------------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Book            | Class     | Book represents a document. A Book object acts as a collection of pages. Pages included in the Book can have identical or differing formats and can use different painters.                                                                 |
| Pageable        | Interface | A Pageable implementation represents a set of pages to be printed. The Pageable object returns the total number of pages in the set as well as the PageFormat and Printable for a specified page. The Book class implements this interface. |
| Printable       | Interface | A page painter must implement the Printable interface. There is only one method in this interface, print( ).                                                                                                                                |
| PrinterGraphics | Interface | The Graphics object implements this interface. PrinterGraphics provides the getPrinterJob( ) method to obtain the printer job that instantiated the print process.                                                                          |

### Pageable interface

The Pageable interface includes following three methods :

| Method name                             | Description                                                                              |
|-----------------------------------------|------------------------------------------------------------------------------------------|
| int getNumberOfPages( )                 | Returns the number of pages in the document.                                             |
| PageFormat getPageFormat(int pageIndex) | Returns the page's PageFormat as specified by pageIndex.                                 |
| Printable getPrintable(int pageIndex).  | Returns the Printable instance responsible for rendering the page specified by pageIndex |

### Printable interface

The Printable interface features one method and two values which are given below :

| Name                                                               | Type   | Description                                                                                                                             |
|--------------------------------------------------------------------|--------|-----------------------------------------------------------------------------------------------------------------------------------------|
| int print(Graphics graphics, PageFormat pageFormat, int pageIndex) | Method | Requests that the graphics handle using the given page format render the specified page.                                                |
| NO_SUCH_PAGE                                                       | Value  | This is a constant. Return this value to indicate that there are no more pages to print.                                                |
| PAGE_EXISTS                                                        | Value  | The print( ) method returns PAGE_EXISTS. It indicates that the page passed as a parameter to print() has been rendered and does exists. |

### PrinterGraphics interface

The PrinterGraphics interface consists of one method :

| Method name                 | Description                                                                                |
|-----------------------------|--------------------------------------------------------------------------------------------|
| PrinterJob getPrinterJob( ) | Returns the PrinterJob for this rendering request and is implemented by the Graphics class |

### Units of measurement

When working with the Graphics2D class, it is important to understand the difference between device space and user space. In the device space, you work in pixels using the resolution of the device. A square of 100 pixels by 100 pixels drawn on a device that has a resolution of 1,024 pixels by 768 pixels will not be the same size as it is when rendered on a device that has a resolution of 1,600 pixels by 1,400 pixels. The reason is simple because the second device features more pixels per inch, the square will appear smaller.

User space, on the other hand, allows us to think in terms of measurement units, regardless of the device's resolution. When you create a Graphics2D object for a given device (printer or screen), a default transform is generated to map the user space to the device space. In user space, the default is set to 72 coordinates per inch. Instead of thinking in terms of pixels, you think in terms of units. A 1-by-1-inch square is 72 units by 72 units. A letter-size page (8.5 by 11 inches) is 612 by 792 points. When using the print API, you must set your mind to work with units because all the classes work in the user space. This is shown later in program 12.14 using getImageableX( ) and getImageableY( ) functions of PageFormat class.

### PageFormat class

The PageFormat consists of 12 methods :

| <b>Method name</b>                   | <b>Description</b>                                                                                                                                                                         |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| double getHeight( )                  | This method returns the page's physical height in points (1 inch = 72 points). If your page measures 8.5 by 11 inches, then the return value will be 792 points, or 11 inches.             |
| double getImageableHeight( )         | This method returns the page's imageable height, which is the height of the print area on which you may draw. See Figure 1 for a graphical view of the imageable area.                     |
| double getImageableWidth( )          | This method returns the page's imageable width — the width of the print area on which you may draw. Figure 1 illustrates a graphical view of the imageable area.                           |
| double getImageableX( )              | This method returns the x origin of the imageable area.                                                                                                                                    |
| double getImageableY( )              | This method returns the imageable area's y origin.                                                                                                                                         |
| double getWidth( )                   | This method returns the page's physical width in points. If you print on letter-sized paper, the width is 8.5 inches, or 612 points.                                                       |
| double getHeight( )                  | This method returns the page's physical height in points. For example, letter-sized paper is 11 inches in height, or 792 points.                                                           |
| double[] getMatrix( )                | This method returns a transformation matrix that translates user space into the requested page orientation. The return value is in the format required by the AffineTransform constructor. |
| int getOrientation( )                | This method returns the orientation of the page as either PORTRAIT or LANDSCAPE.                                                                                                           |
| void setOrientation(int orientation) | This method sets the orientation of the page, using the constants PORTRAIT and LANDSCAPE.                                                                                                  |
| Paper getPaper( )                    | This method returns the Paper object associated with the page format. Refer to the previous section for a description of the Paper class.                                                  |
| void setPaper(Paper paper)           | This method sets the Paper object that will be used by the PageFormat class. PageFormat must have access to the physical page characteristics to complete this task.                       |

This concludes the description of the page classes. The next class that we will study is the PrinterJob.

### PrinterJob class

The PrinterJob class controls the printing process. It can both instantiate and control a print job. Below you will find a definition of the class:

| <b>Method name</b>                                               | <b>Description</b>                                                                                                                                                                                                                                                                                                                                                                                                                               |
|------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| abstract void cancel( )                                          | This method cancels the current print job. You can validate the cancellation with the isCancel( ) method.                                                                                                                                                                                                                                                                                                                                        |
| abstract boolean isCancelled( )                                  | This method returns true if the job is cancelled.                                                                                                                                                                                                                                                                                                                                                                                                |
| PageFormat defaultPage( )                                        | This method returns the default page format for the PrinterJob.                                                                                                                                                                                                                                                                                                                                                                                  |
| abstract PageFormat defaultPage(PageFormat page)                 | This method clones the PageFormat passed in parameters and modifies the clone to create the default PageFormat.                                                                                                                                                                                                                                                                                                                                  |
| abstract int getCopies( )                                        | This method returns the number of copies that the print job will print.                                                                                                                                                                                                                                                                                                                                                                          |
| abstract void setCopies(int copies)                              | This method sets the number of copies that the job will print. Note that if you show a print dialog box, users can alter the number of copies (see the pageDialog method).                                                                                                                                                                                                                                                                       |
| abstract String getJobName( )                                    | This method returns the job name.                                                                                                                                                                                                                                                                                                                                                                                                                |
| static PrinterJob getPrinterJob( )                               | This method creates and returns a new PrinterJob.                                                                                                                                                                                                                                                                                                                                                                                                |
| abstract String getUsername( )                                   | This method returns the user name associated with the print job.                                                                                                                                                                                                                                                                                                                                                                                 |
| abstract PageFormat pageDialog(PageFormat page)                  | This method displays a dialog that allows the user to modify the PageFormat. The PageFormat, passed in parameters, sets the fields of the dialog. If the user cancels the dialog, then the original PageFormat will be returned. But if the user accepts the parameters, then a new PageFormat will be created and returned. Since it will not show the same parameters on all operating systems, you must be careful when using the pageDialog. |
| abstract void setPageable(Pageable document)                     | This method queries the document to obtain the total number of pages. The Pageable will also return the PageFormat and the Printable object for each page. See the definition of the Pageable interface for more information.                                                                                                                                                                                                                    |
| abstract void setPrintable(Printable painter)                    | This method sets the Painter object that will render the pages to be printed. A Painter object is an object that implements the Printable class and its print( ) method.                                                                                                                                                                                                                                                                         |
| abstract void setPrintable(Printable painter, PageFormat format) | This method completes the same tasks as abstract void setPrintable(Printable painter), except that you supply the PageFormat that the Painter will use. As indicated in the definition of the Printable interface, the print( ) method passes a PageFormat object as the first parameter.                                                                                                                                                        |
| abstract void print( )                                           | This method prints the document. It actually calls the print() method of the Painter previously assigned to this print job.                                                                                                                                                                                                                                                                                                                      |
| abstract void setJobName(String jobName)                         | This method sets the name of the print job.                                                                                                                                                                                                                                                                                                                                                                                                      |
| abstract boolean printDialog( )                                  | This method displays a print dialog box that allows the user to change the print parameters. Note that this interaction's result will not be returned to your program. Instead, it will be passed to the peer operating system.                                                                                                                                                                                                                  |
| abstract PageFormat validatePage(PageFormat page)                | This method will validate the PageFormat passed in parameters. If the printer cannot use the PageFormat that you supplied, then a new one that conforms to the printer will be returned.                                                                                                                                                                                                                                                         |

### The Printing Sequence

After discussing a few of the basics needed to print in Java, let us now talk about actual printing sequence in Java. In Java, printing task is usually carried out in *two parts* :

- ◆ **Job control** — Creating a print job, associating it with a printer, specifying the number of copies, and user print dialog interaction.
- ◆ **Page Imaging** — Drawing content to a page, and managing content that spans pages (pagination).

Following lines illustrate this process.

### 1. Creating Print Job

In order to print in Java, firstly create the printer job. The class representing a printer job and most other related classes is located in the `java.awt.print` package. That i.e., you need to write following code to create the print job.

```
import java.awt.print.*; // import all classes of java.awt.print package
PrinterJob printJob = PrinterJob.getPrinterJob();
```

See, to create the print job, you need to create an object or `PrinterJob` class by invoking its `getPrinterJob()` method via the job object(object of `PrinterJob` type).

### 2. Render Content to Page for Printing

Next provide code that renders the content to the page by implementing the `Printable` interface.

```
class HelloWorldPrinter implements Printable
{ ...
 ...
 printjob.setPrintable(new HelloWorldPrinter());
```

*Notice, Printable interface  
is being implemented*

*Invoke setPrintable() to render  
the content to page*

Or you could set the `PageFormat` along with the Painter :

```
printJob.setPrintable (Painter, pageFormat);
```

Finally, the Painter object must implement the `print()` method :

```
public int print (Graphics g, PageFormat pageFormat, int page)
```

Here the first parameter is the graphics handle that you will use to render the page, the `PageFormat` is the format that will be used for the current page, and the last parameter is the page number that must be rendered. That's all there is to it — for simple printing, that is.

### 3. Display Print Dialog

An application typically displays a print dialog so that the user can adjust various options such as number of copies, page orientation, or the destination printer. This is done by invoking `printDialog()` method via the `printjob` object( object of `PrinterJob` type).

```
boolean doPrint = printjob.printDialog();
```

This dialog appears until the user either approves or cancels printing. The `doPrint` variable will be true if the user gave a command to go ahead and print. If the `doPrint` variable is false, the user cancelled the print job. Since displaying the dialog at all is optional, the returned value is purely informational.

If the `doPrint` variable is true, then the application will request that the job be printed by calling the `PrinterJob.print` method.

```
if (doPrint) {
 try {
```

```

 job.print();
 } catch (PrinterException e) {
 /* The job did not successfully complete */
 }
}

```

The `PrinterException` will be thrown if there is problem sending the job to the printer. However, since the `PrinterJob.print` method returns as soon as the job is sent to the printer, the user application cannot detect paper jams or paper out problems. This job control boilerplate is sufficient for basic printing uses.

The `Printable` interface has only one method:

```

public int print(Graphics graphics, PageFormat pf, int page)
throws PrinterException;

```

The `PageFormat` class describes the page orientation (portrait or landscape) and its size and imageable area in units of 1/72nd of an inch. *Imageable* area accounts for the margin limits of most printers (hardware margin). The *imageable* area is the space inside these margins, and in practice it is often further limited to leave space for headers or footers.

A `page` parameter is the zero-based page number that will be rendered.

### Print your first page

Recall that the print system in Java has two distinct models: the *Printable* and the *Pageable*. Though `Printable` can print simple documents, it features several limitations, the major one being that all pages must share the same format. The *Pageable* model, on the other hand, offers much more flexibility. Used in conjunction with the `Book` class, it can create multipage documents, with each page formatted differently. Listing 1 shows how to print using the `Printable` model. We're however not going in details of how to print with other models.

#### Program 12.14

Write a Program to print  $\frac{1}{2}$  inch by  $\frac{1}{2}$  inch grid.

```

import java.awt.*;
import java.awt.geom.*;
import java.awt.print.*;
public class PrintExample1 implements Printable {
 private final double INCH = 72; // Private instances declarations
 public PrintExample1() { // Constructor:
 PrinterJob printJob = PrinterJob.getPrinterJob(); //Create a printerJob object
 // Set the printable class to this one since we are implementing the Printable interface
 printJob.setPrintable(this);
 // Show a print dialog to the user. If the user clicks the print button, then print,
 // otherwise cancel the print job
 if (printJob.printDialog()) {
 try {
 printJob.print();
 }
 }
 }
}

```

```

 } catch (Exception PrintException) {
 PrintException.printStackTrace();
 }
 }

 / * Method: print <p>
 * This method is responsible for rendering a page using the provided parameters.
 * The result will be a grid where each cell will be half an inch by half an inch. */
 public int print (Graphics g, PageFormat pageFormat, int page) {
 int i;
 Graphics2D g2d;
 Line2D.Double line = new Line2D.Double();
 if (page == 0) { // Validate the page number, we only print the first page
 g2d = (Graphics2D) g; // Create a graphic2D object and set the default parameters
 g2d.setColor (Color.black);
 // Translate the origin to be (0,0)
 g2d.translate (pageFormat.getImageableX(), pageFormat.getImageableY());
 // Print the vertical lines
 for (i = 0; i < pageFormat.getWidth(); i += INCH / 2) {
 line.setLine (i, 0, i, pageFormat.getHeight());
 g2d.draw (line);
 }
 // Print the horizontal lines
 for (i = 0; i < pageFormat.getHeight(); i += INCH / 2) {
 line.setLine (0, i, pageFormat.getWidth(), i);
 g2d.draw (line);
 }
 }
 return (PAGE_EXISTS);
 }
 else
 return (NO_SUCH_PAGE);
}
} //PrintExample1

```

Above program will print a half-inch by half-inch grid using the default margin setting, usually one inch for the top, bottom, left, and right margins. Please note that if you try to execute this example with Java 1.2, the margins will not fit; the left margin will be slightly larger than one inch. A bug in Java 1.2's print API causes this glitch.

Now, let's go through the steps required for printing with the `Printable` model, once again :

1. Create a `PrinterJob` object. This object controls the print process by displaying page and print dialogs, and initiating the print action.
2. Display the proper dialogs, either print or page dialogs.
3. Create a class that implements the `Printable` interface's `print()` method.
4. Validate the page number to be rendered.
5. Render your page using the `Graphics` parameter.
6. If the page renders, return the `PAGE_EXISTS` value; if the page does not render, return the `NO_SUCH_PAGE` value.

## L et Us Revise

- ❖ Data that need to be stored permanently must be stored in files.
- ❖ There are two types of streams supported in Java : byte oriented streams and character streams.
- ❖ The byte stream classes are categorized into Input Stream Classes and Output Stream Classes.
- ❖ The character stream classes are categorized into Reader and Writer classes.
- ❖ To perform output on text files, following steps are performed :
  - Create `FileWriter` stream for a file to be written onto.
  - Connect it to `BufferedWriter`.
  - Connect `BufferedWriter` to `PrintWriter`.
  - Write text on file using `print()`/`println()`.
  - Once through, close the streams using `close()`.
- ❖ To perform input on text files, following steps are performed :
  - Create `FileReader` Stream for a file to be read.
  - Connect it to `BufferedReader`.
  - Read text from file using `readLine()`.
  - Once through close the file streams `close()`.
- ❖ To perform output on binary files, following steps are performed :
  - Create a `FileOutputStream` and link it with the file.
  - Connect it to `DataOutputStream`.
  - Write data on the file using any of the `write()` functions.
  - Once through close the streams.
- ❖ To perform input on binary files, following steps are performed :
  - Create `FileInputStream` and link it with file.
  - Connect it to a `DataInputStream`.
  - Read data using any of the `read()` functions.
- ❖ A string tokenizer can identify and parse (segregate) tokens in a string.
- ❖ A stream tokenizer can identify and parse tokens in a stream.
- ❖ Scanner class can also be used for obtaining input.
- ❖ `Printable` and `Pageable` are the two ways of printing in Java.
- ❖ `Printable` is simpler way of printing whereas `Pageable` is more flexible way of printing that offers many features.

## S olved Problems

1. What is the difference between text files and binary files ?

**Solution.** In text files, data are stored as per a specific character encoding scheme e.g., ASCII text or Unicode text is stored.

In binary files, data are stored in the form of bytes that are in machine readable form.

2. Which streams are used for performing IO in (a) text files ? (b) binary files ?

**Solution.**

(i) `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`, `PrintWriter`.

(ii) `FileInputStream`, `FileOutputStream`, `DataInputStream`, `DataOutputStream`.

3. How many bytes does the following code write to file dest ?

```
try { FileOutputStream fos = new FileOutputStream("dest") ;
 DataOutputStream dos = new DataOutputStream(fos) ;
 dos.writeInt(3) ;
 dos.writeDouble(0.0001) ;
 dos.close() ;
 fos.close() ;
 }
catch (IOException e) {}
```

(a) 2      (b) 8      (c) 12      (d) 16

**Solution.** (c) The `writeInt()` call writes out an int, which is 4 bytes long ; the `writeDouble()` call writes out a double, which is 8 bytes long, for a total of 12 bytes.

4. A file is created with the following code :

```
FileOutputStream fos = new FileOutputStream("datafile") ;
DataOutputStream dos = new DataOutputStream(fos) ;
for(int i = 0 ; i < 500 ; i++)
 dos.writeInt(i) ;
```

You would like to write code to read back the data from this file. Which solutions listed below will work ?

- (a) Construct a `FileInputStream`, passing the name of the file. Onto the `FileInputStream`, chain a `DataInputStream`, and call it `readInt()` method.
- (b) Construct a `FileReader`, passing the name of the file. Call the file reader's `readInt()` method.
- (c) Construct a `PipedInputStream`, passing the name of the file. Call the piped input stream's `readInt()` method.
- (d) Construct a `FileReader`, passing the name of the file. Onto the `FileReader`, chain a `DataInputStream`, and call it `readInt()` method.

**Solution.** (a), (d).

- (a) chains a data input stream onto a file input stream.
- (b) fails because the `FileReader` class has no `readInt()` method ; readers and writers only handle text.
- (c) fails because the `PipedInputStream` class has nothing to do with file I/O. (Piped input and output streams are used in inter-thread communication).
- (d) fails because you cannot chain a data input stream onto a file reader. Readers read `chars`, and input streams handle `bytes`.

5. Which of the following statements are true ? Choose all correct answers.

- (a) You can directly read text from a `FileReader`.
- (b) You can directly write text to a `FileWriter`.
- (c) You can directly read ints and floats from a `FileReader`.
- (d) You can directly write longs and shorts to a `FileWriter`.

**Solution.** (a), (b). Only (a) and (b) are true. Readers and writers are exclusively for text.

6. Copy contents of a text file on to another text file.

**Solution.**

```
import java.io.* ;
public class CopyFile {
```

```

public static void main(String[] args) {
 try {
 FileWriter fw = new FileWriter("Copy.txt") ;
 BufferedWriter bw = new BufferedWriter(fw) ;
 PrintWriter outFile = new PrintWriter(bw) ;
 FileReader file = new FileReader("names.txt") ;
 BufferedReader fileInput = new BufferedReader(file) ;
 String text ;
 // Read file and output
 while((text = fileInput.readLine()) != null) {
 outFile.println(text) ;
 }
 System.out.println("File successfully copied!!") ;
 fileInput.close() ; // Close file
 outFile.close() ;
 }
 catch(IOException e) {
 System.err.println(e) ;
 }
}
}

```

7. Rollno and marks of some students are stored in file stu.dat. Create a no Toppers.dat that contains data of only those students who have scored more than 75% marks.

**Solution.**

```

import java.io.* ;
public class ExtractData {
 public static void main(String[] args) {
 try {
 int rno ; float marks ;
 FileOutputStream fw = new FileOutputStream("Toppers.dat") ;
 DataOutputStream dw = new DataOutputStream(fw) ;
 FileInputStream fr = new FileInputStream("Stu.dat") ;
 DataInputStream dr = new DataInputStream(fr) ;
 boolean EOF = false ;
 while(!EOF) {
 try {
 rno = dr.readInt() ;
 marks = dr.readFloat() ;
 if(marks > 75) {
 dw.writeInt(rno) ;
 dw.writeFloat(marks) ;
 System.out.println(rno + " is topper with marks" + marks) ;
 }
 } // end of inner try
 catch (EOFException e1) {
 System.out.println("end of file") ;
 EOF = true ;
 }
 }
 }
 }
}

```

```
 catch (IOException e) {
 System.err.println(e) ;
 }
 } // end of while
}
dw.close() ;
fw.close() ;
System.out.println("\nTopper file successfully created!!!") ;
} // end of outer try block
catch (FileNotFoundException e) {
 System.out.println("File not found") ;
}
catch (IOException e) {
 System.err.println(e) ;
}
```

## Glossary

**File** A bunch of bytes stored on some storage device

**Streams** Controlled flows of data from one source to another.

**Tokenizer Classes** Classes that provide functionality to parse strings or tokens.

**Print Job** Unit of work to be run on a printer and which can consist of multiple pages.

## Assignments

**TYPE A : VERY SHORT/SHORT ANSWER QUESTIONS**

1. What is file ?
  2. Which library file of Java provides facilities for file input/output operations ?
  3. A Java stream is
    - (i) the flow of control through a function
    - (ii) a flow of data from one place to another
    - (iii) associated with a particular class
    - (iv) a file.
  4. Name three stream classes commonly used for character oriented I/O.
  5. What is the role of `InputStream` class ?
  6. What is the role of `Reader` class ?
  7. What is the role of `OutputStream` class ?
  8. What is the role of `Writer` class ?
  9. Name the most commonly used classes for handling Byte oriented IO.
  10. Create a `DataInputStream` for the file named "student.dat"

## OPERATIONS ON FILES

11. Can we open an existing file for writing ? If not, why ?
12. What are input and output streams ?
13. Write statements for how integer type value is read from the keyboard interactively.
14. Why are streams needed ?
15. Write statements to create data streams for the following operations :
  - (a) Reading primitive data from a file
  - (b) Writing primitive data to a file
16. What for can you use Scanner class ?
17. How are Pageable and Printable printing mechanisms different ?

**TYPE B : LONG ANSWER QUESTIONS**

1. What are input and output streams ? What is the significance of java.io package file ?
2. Discuss important stream classes defined inside java.io package file.
3. Describe the various classes available for file operations.
4. When a file is opened for output, what happens when
  - (i) the mentioned file does not exist
  - (ii) the mentioned file does exist.
5. Distinguish between
  - (a) InputStream and Reader classes
  - (b) OutputStream and Writer classes
6. What is meant by initializing a file stream object ? What are the ways of doing it ? Give example code for each of them.
7. State the steps involved in creating a disk file.
8. Write a code snippet that will create an object called filout for writing, associate it with the filename STRS. The code should keep on writing strings to it as long as the user wants.
9. What are the advantages of saving data in (i) binary form (ii) text form ?
10. When do you think text files should be preferred over binary files ?
11. Write a program that reads a text file and creates another file that is identical except that every sequence of consecutive blank spaces is replaced by a single space.
12. Write a program that copies one file to another. Has the program to take the file names from the users ? Has the program to refuse copy if there already is a file having the target name ?
13. Write a program that appends the contents of one file to another. Have the program take the filenames from the user ?
14. Write a program that reads characters from the keyboard one by one. All lowercase characters get stored inside the file LOWER, all uppercase characters get stored inside the file UPPER and all other characters get stored inside file OTHERS.
15. Write a program to search the name and address of person having age more than 30 in the file of persons.dat.
16. Write a program that counts the number of characters in a file.
17. Modify the above program so that it also counts the number of words, and lines in the file.
18. Write a program to create a sequential file that could store details about five products. Details include product code, cost and number of items available and are provided through the keyboard.
19. Write a program to read the file created in question 18 and compute and print the total value of all the five products.
20. Write a short program to print a square of '\*'s using Java Printing API.

## *Trends in Computing and Ethical Issues*

### **13.1 Introduction**

The rapid growth of technology is not at all surprising but what surprises is the rate the technology is growing and evolving, impacting our lives in a way no one could have imagined some decades ago. Today, you can listen to news or play your favourite music without getting up from your place, by just giving spoken instructions. Based on your preferences or search history, your online shopping store recommends products for you and you are surprised with their thoughtful matching choices, no ? All this and much more, is possible because of modern age technologies.

In this chapter, we shall talk about some recent trends in computing along with cyber security threats and related issues and about intellectual property and related issues.

### **In This Chapter**

- 
- 13.1 Introduction
  - 13.2 Trends in Computing
  - 13.3 Cyber Security
  - 13.4 Major Ethical Issues
  - 13.5 Open Source and Its Philosophy
  - 13.6 Netiquette
  - 13.7 Email Etiquette

## 13.2 Trends in Computing

Field of computer technology is ever evolving field. From the bulky room size machines to today's smartphones fitting on a palm, the growth of computers is phenomenal (☺). Throughout the development of the computer, various people have made contributions affecting the way a computer works. In this section, let us talk about some recent trends in computing technology.

### 13.2.1 Artificial Intelligence

When computers were devised, despite being very fast and accurate machines, they could not learn on their own or imitate humans. But, things have changed now. A special branch of computer science, *Artificial Intelligence* has made it possible.

Artificial Intelligence (AI) basically refers to the ability of a machine or a computer program to think and learn. In simple words, field of AI revolves around bringing out technologies that help build machines that can think, act, and learn like humans.

An AI based program and technology should bring out these things :

- ❖ Firstly, it should be able to mimic human thought process and behavior e.g., learning from mistakes, catching up with new ideas, learning new things from new exposure, experiences etc.
- ❖ Secondly, it should act in a human-like way—*intelligent, rational, and ethical*, i.e., it should be able to take right decisions in ethical ways.

**Note** Field of AI revolves around bringing out technologies that help build machines that can think, act, and learn like humans.

Most famous example of AI today is social humanoid robot Sophia, who has been awarded citizenship of Saudi Arabia. Other common examples of AI today are :

- ❖ Siri or Alexa — the personal assistants that have already become the new normal for thousands of people around the globe.
- ❖ smart home devices like Google's NEST,
- ❖ self-driving cars like those produced by Tesla,
- ❖ online games like *Alien : Isolation*.

**Artificial Intelligence (AI)** refers to the ability of a machine or a computer program to think, learn and evolve.

It is predicted that in coming 5 to 10 years AI will grow tremendously. AI based machines would outperform humans in tasks such as *translating languages, writing school essays, driving trucks etc.* However, more complicated tasks like *operating in place of a surgeon or coming out with heart-touching emotional books or bestsellers etc.* will take machines much more time to learn. AI is expected to master these skills in coming decades.

### 13.2.2 Internet of Things (IoT)

You must have read stories like : "a person was able to monitor his home through his smartphone even while he himself was sitting in another country"; "scientists were able to monitor the progress of a whale who was operated upon and a chip was inserted in her body"; "a car sensor alerted the car owner

*about low air pressure in tires in time*" and so on. All these are nothing but some examples of applications of IoT. But what is IoT? Have we discussed that already? No, in a moment, we are going to do this ☺.

The IoT (Internet of Things) is a new age technology that allows computing devices (*devices that can be programmed and can connect to Internet such as smart home appliances like smart refrigerators or smart air conditioners, a smart heart monitor chip etc.*), to transfer data over a network like Internet without requiring human-to-human or human-to-computer interaction.

The IoT (Internet of Things) is a new age technology that allows computing devices to transfer data over a network like Internet without requiring human-to-human or human-to-computer interaction.

Practical applications of IoT technology can be found in many fields/areas today, such as :

- ◆ **Health and Fitness.** IoT smart gadgets like *Fitbit, Jawbone, Nike and Misfit* etc., that monitor your heart rate, blood pressure etc. and take action accordingly such as sending emergency messages or updating daily fitness log or contacting appointed doctor etc.
- ◆ **Home Security.** There are many home safety and security devices for everyone that enable *video surveillance, motion, temperature and air quality control* to help you protect your family and your home when you're not around.
- ◆ **Transport.** Driverless cars can 'not only' drive on road without drivers but also can be in touch with servers all time.
- ◆ **Shopping.** There are smart refrigerators nowadays that can order for grocery items as soon their quantity in fridge goes below a set level.
- ◆ **Smart Cities.** The IoT technology is main enabler of making 'Smart city' a reality. *Smart surveillance, automated transportation, smarter energy management systems, water distribution, urban security and environmental monitoring etc.*, are examples of Internet of Things applications for smart cities.

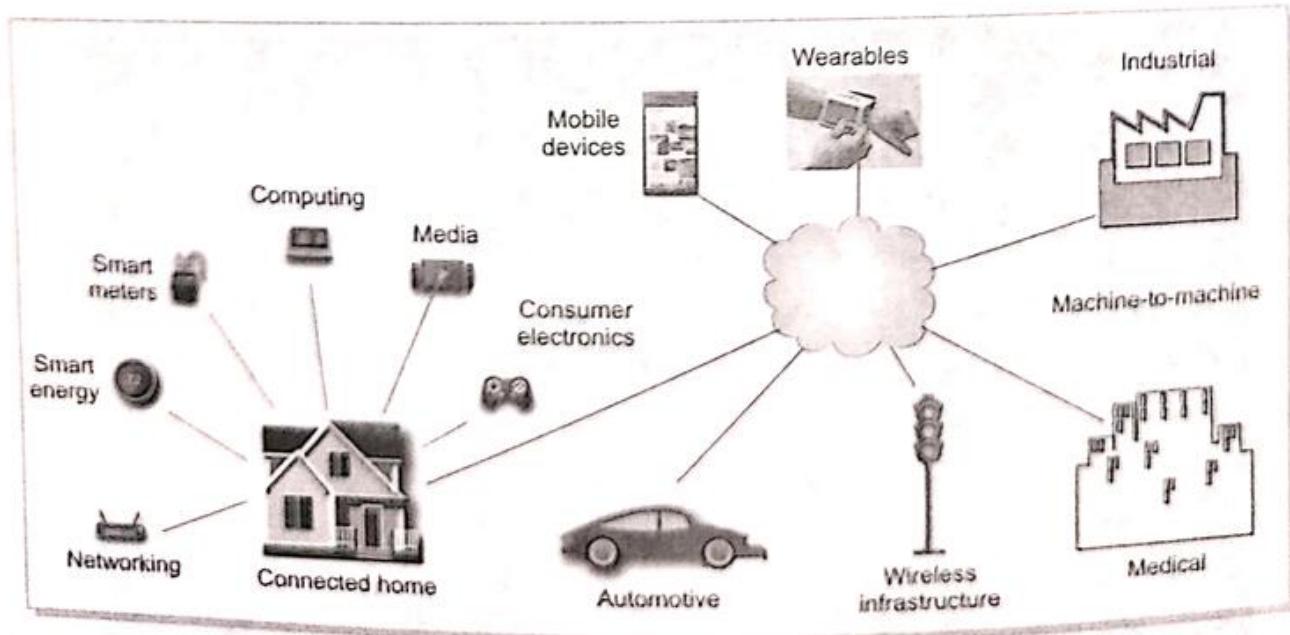


Fig. 13.1

### 13.2.3 Virtual Reality (VR)

You must have seen 3D movies. Don't you feel that everything is happening in front of you; everything is real. For example, if in a 3D movie, a shot has train coming fast straight unto you, chances are that you may scream as it will feel real to you. Here things appear real but everything is pre-scripted and pre-shot, you cannot alter things interactively.

Virtual Reality is the next level of this. Here a user experiences a real world like environment in a virtual environment with the help of some digital equipment but he/she can change the look and feel of it interactively too. Please note that for virtual reality a 3D virtual environment is simulated which is generated and reproduced by the CPU of a specially designed VR device.

Virtual Reality makes it possible for users to interact with a virtual environment with multiple senses (as many senses as possible), including *sight, hearing, touch*, and sometimes even *smell* and *taste*. This is called **sensory synchronicity**.

A VR experience is made possible generally with devices like :

- ❖ A **VR headset or helmet** that mounts on head of the user and produces a 3d imagery based environment that appears real and in which the user can interactively participate.
- ❖ An **instrumented and sensory VR glove(s)** is a glove laden with sensors that digitally translate every movement of your hand and reflect that on the virtual 3D environment.
- ❖ An **instrumented and sensory VR bodysuit** is body fitted with sensors all over the body suit that are able to translate every movement of body digitally and reflect them over a virtually generated environment.

**Virtual Reality (VR)** is a technology that allows people to experience and interact in a 3D virtual environment that appears and feels like a real environment with the use of an electronic equipment.



Consider some examples of virtual reality :

- ◆ You are a Ski player and want to participate in coming Ski Race. But you live in a place where there is not enough snow and matching terrain where you can practice. A company in your city helps you practice this with its VR setup and you get ample practice.
- ◆ Modern military training camps enable real fight and combat situations through VR environment.
- ◆ Pilots can learn and practice flying of the aeroplanes through VR environments.
- ◆ A surgery student can practice multiple surgeries in a VR environment before actually carrying out a real surgery.

and many more such applications.

**Note** Common VR devices popular today are Oculus Rift, HTC Vive, Gear VR, PlayStation VR etc.

#### 13.2.4 Augmented Reality (AR)

Have you heard of the mobile game 'Pokémon Go'? You simply walk around with the game app open on your phone, which will buzz when Pokémon are nearby, which you can catch using the *Pokémon Go* game app. But have you wondered – how come the Pokémon (virtual, scripted, animation characters) are nearby in your surroundings? Well, the answer to this is augmented reality wherein the computer generated 3D imagery in the form of Pokémon, is superimposed on your real world.

Augmented reality is a new age technology that expands our physical real world by adding layers of digital information onto it i.e., by adding digitally generated images/information etc., on it and thus transforms our view of our surroundings.



The **Augmented Reality (AR)** is a technology that transforms the view of physical real-world environment with superimposed computer-generated images, thus changing the perception of reality.

#### Augmented Reality Devices

Many modern devices already support Augmented reality — digital devices that can support sensors, cameras, accelerometer, gyroscope, digital compass, GPS, CPU, projected displays etc.

Devices suitable for *Augmented reality* can be any one of these :

- ◆ *Mobile devices* (smartphones and tablets)
- ◆ *Special AR devices* such as *head-up displays (HUD)*, designed primarily and solely for augmented reality experiences.
- ◆ *AR glasses* (or smart glasses) such as Google Glasses, Meta 2 Glasses, Laster See-Thru, Laforgo AR eyewear, etc.
- ◆ *AR contact lenses* (or smart lenses).
- ◆ *Virtual retinal displays (VRD)*, creating images by projecting laser light into the human eye.

### Augmented Reality vs. Virtual Reality

While both augmented reality and virtual reality alter our view, augmented reality superimposes the generated imagery and information on our existing physical world and what we view is a mix of real world and the digitally generated imagery/information. Virtual reality, on the other hand, replaces the physical world around us with a virtual world altogether.

Table 13.1 Augmented Reality (AR) vs Virtual Reality (VR)

| Augmented Reality (AR)                                                         | Virtual Reality (VR)                                                                 |
|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| Augmented reality is a mix of the real world and the virtual world.            | Virtual reality creates an entire virtual world.                                     |
| It lets people interact with both worlds and distinguish clearly between both. | In this case, it is hard to differentiate between what is real and what is not real. |
| This is generally achieved by holding a smartphone in front of you.            | This is generally achieved by wearing a helmet or goggles.                           |

Following Fig. 13.2 describes the difference between AR and VR.

**Note** In Virtual reality, your screen becomes your world while in Augmented reality, the world is your screen.

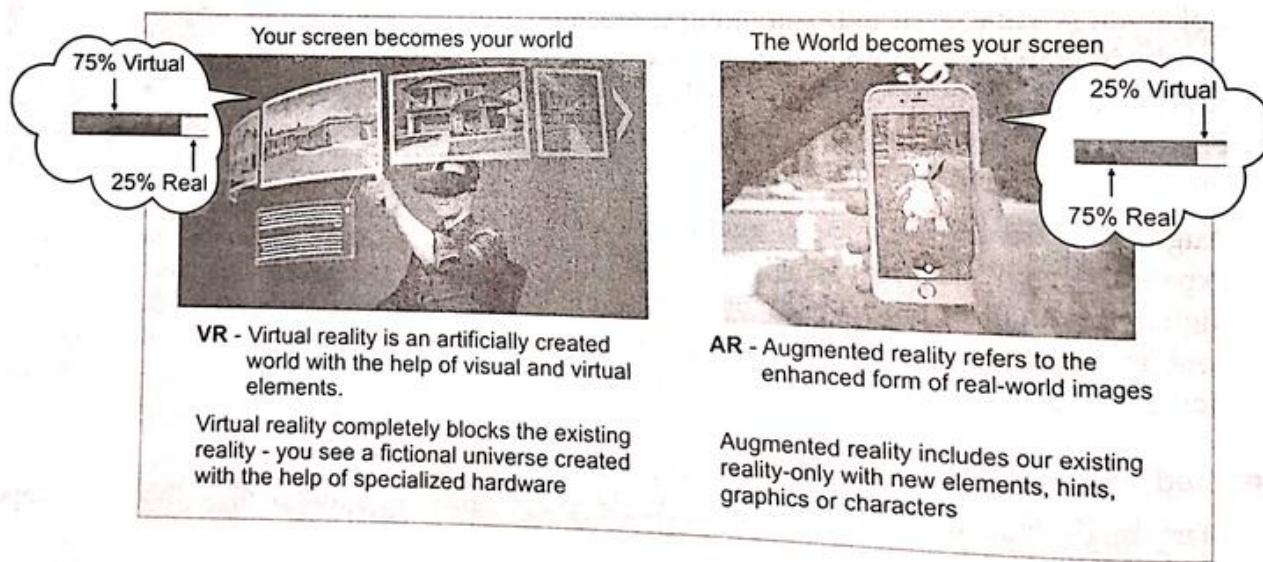


Fig. 13.2

### 13.3 Cyber Security

Network security is important because globally people, businesses, corporates etc. are dependent on computer network for storing, accessing and providing business information. This is the reason that various threats to network security are studied and a counter plan is prepared to secure the networks.

#### 13.3.1 Threats to Computer Security

A threat is a potential violation of security. When a *threat* is actually executed, it becomes an *attack*. Those who execute such actions, or cause them to be executed are called *attackers*.

Some common threats the average computer user faces every day are being given below.

### Computer Viruses

Computer viruses are malicious codes/programs that cause damage to data and files on a system. Viruses can attack any part of a computer's software such as boot block, operating system, system areas, files and application-program-macros. Two other similar programs also cause virus like effects. These are :

- (a) **Worms.** A worm is a self-replicating program which eats up the entire disk space or memory. A worm keeps on creating its copies until all the disk space or memory is filled.
- (b) **Trojan Horses.** A trojan horse is a program that appears harmless (such as a text editor or a utility program) but actually performs malicious functions such as deleting or damaging files.

**Damage caused by Viruses.** *Viruses can do the following if left unchecked :*

- ❖ Damage or delete files.
- ❖ Slow down your computer.
- ❖ Invade your e-mail program.

### Spyware

Spyware is a software which is installed on your computer to spy on your activities and report this data to people willing to pay for it. It tracks the user's behaviour and reports information back to a central source. These are used to spy on some one either for legal or illegal purpose.

**Damage caused by Spyware.** *Spyware can act like a peeping tom or, at worse, a geeky thief. For example, it :*

- ❖ Compromises your data, computing habits, and identity.
- ❖ Alters PC settings.
- ❖ Slows down your PC.

### Adware

These are the programs that deliver unwanted ads to your computer (generally in Pop-Ups form). They consume your network bandwidth. Adware is similar to spyware—however, it may be installed with your consent. So it is advised that you thoroughly read installation agreements before you allow installation of a software.

**Damage caused by Adware.** *Adware comes complete with the following disadvantages :*

- ❖ Adware tracks information just like spyware.
- ❖ Displays arrays of annoying advertising.
- ❖ Slows down your PC.

### Spamming

Spamming refers to the sending of bulk-mail by an identified or unidentified source. In non-malicious form, bulk-advertising mail is sent to many accounts. In malicious form (e.g., e-mail bombing), the attacker keeps on sending bulk mail until the mail-server runs out of disk space.

**Damage caused by Spam**

- ❖ Spam reduces productivity.
- ❖ Spam eats up your time.
- ❖ Spam can lead to worse things.

## PC Intrusion

Every PC (personal computer) connected to the Internet is a potential target for hackers. Computers are under constant attack from cyber vandals. PC Intrusion can occur in any of the following form.

- (i) **Sweeper Attack.** This is another malicious program used by hackers. It sweeps i.e., deletes all the data from the system.
- (ii) **Denial of Services.** This type of attack eats up all the resources of a system and the system or applications come to a halt. Example of such an attack is flooding a system with junk mail.
- (iii) **Password Guessing.** Most hackers crack or guess passwords of system accounts and gain entry into remote computer systems. And then they use it for causing damages in one or another form.
- (iv) **Snooping.** Snooping refers to opening and looking through files in unauthorized manner. Snooping may involve many type of things such as *gaining access of data in unauthorized way, casually observing someone else's email or monitoring activity of someone else's computer through a sophisticated snooping software*. It may involve – monitoring of keystrokes pressed, capturing of passwords and login information and interception of email and other private communicate and data transmission.
- (v) **Eavesdropping.** Eavesdropping is also similar to snooping. When some one listens to a conversation that they are not part of, it is eavesdropping. Formally you can say that the intentional interception of someone else's data (such as email, login-id, password, credit card info etc.) as it passes through a user's computer to server or vice-versa is called eavesdropping.

## Phishing

It is the criminally fraudulent process of attempting to acquire sensitive information such as usernames, passwords, credit card information, account data etc. In Phishing, an imposter uses an authentic looking e-mail or website to trick recipients into giving out sensitive personal information. For instance, you may receive an e-mail from your bank (which appears genuine to you) asking to update your information online by clicking at a specified link. Though it appears genuine, you may be taken to a fraudulent site where all your sensitive information is obtained and later used for cyber-crimes and frauds.

### 13.3.2 Cyber Security Measures

The combination of identification, authentication and authorization can control access to a system. This combination is very useful especially in network security. Various techniques used for network security are given below :

#### Active Protection

##### ❖ Authorization

Asking the user a legal login-id performs authorization. If the user is able to provide a legal login-id, he/she is considered an *authorized user*.

##### ❖ Authentication

Authentication is also termed as **password-protection** as the authorized user is asked to provide a valid password, and if he/she is able to do this, he/she is considered to be an *authentic user*.

#### ❖ Firewall

A system designed to prevent unauthorized access to or from a private network is called **Firewall**. Firewalls are a mechanism to prevent unauthorized Internet users from accessing private networks connected to the Internet, especially intranets.

A system designed to prevent unauthorized access to or from a private network is called **Firewall**.

#### ❖ Intrusion Detection

Intrusion detection is the art and science of sensing when a system or network is being used inappropriately or without authorization. An intrusion-detection system (IDS) monitors system and network resources and activities and, using information gathered from these sources, notifies the authorities when it identifies a possible intrusion.

### Preventive Measures

#### ❖ Implement proper security policy for your organization

A security policy is a formal statement of the rules that people who are given access to an organization's technology and information assets must abide. The policy communicates the security goals to all of the users, the administrators, and the managers.

A good security policy must :

- Be able to be enforced with security tools, where appropriate, and with sanctions, where actual prevention is not technically feasible.
- Clearly define the areas of responsibility for the users, the administrators, and the managers.
- Be flexible to the changing environment of a computer network since it is a living document.

#### ❖ Use proper file access permissions when sharing files on a network.

File access permissions refer to privileges that allow a user to read, write or execute a file.

- If a user has **Read permission** for a file, he/she can view and read the file. Shown as r.
- If a user has **Write permission** for a file, he/she can edit and write into the file. Shown as w.
- If a user has **Execute permission** for a file, he/she can execute the file. Shown as x.

#### ❖ Disconnect from the Internet when away.

Using "always on" Internet connections such cable and DSL increases your chances of some infections and intrusions as your PC is always connected to the Internet. This doesn't mean you should switch back to dial-up Internet — however, you may want to disconnect from your "always on" connection when you don't plan on using it for a long period of time.

#### ❖ Never click on a link you received in your email.

Even if the link appears genuine, do not click on it. If you have to visit this site, then you type the URL of the site yourself in the address bar. This way you can avoid phishing attacks.

### 13.3.3 Online Privacy

Now that so much of normal life revolves around the Internet, the privacy of each and every one of us is at risk. Advertisers, service providers, and governments all around the world are increasingly interested in tracking every single movement we make online. Whatever we do online such as search for something, visit sites, access social networking sites, posting something, making payments online etc., everything is being tracked and this information is being used for legitimate as well as illegitimate purposes.

'How to safeguard one's privacy online', requires you to know and follow certain rules while working online.

#### Practices to Ensure Confidentiality of Information

Computers, networking, Internet – these all are inevitable ; you cannot avoid these as these are modern age tools. But you surely can follow certain practices to safeguard your data and ensure its confidentiality.

Best practices used to ensure confidentiality are as follows :

##### 1. Use firewall wherever possible

Your system must be secured such that only authentic users can connect to it. Firewall is one very good solution for this. Firewall is a program that monitors all communications and traps all illicit packets. Most operating systems now come with a firewall preinstalled. However, some, such as the Windows firewall, only block suspect incoming communications, leaving completely open access to the Internet from your machine.

Thus, it is recommended that you install a firewall that can monitor both incoming and outgoing communication and traps the illicit ones.

##### 2. Control Browser settings to block tracking

You already know that websites can track your surfing on their site by IP address and related system information, including system names and Internet network addresses that often uniquely identify your computer. Search engines generally record your queries together with your computer identification, building up a profile of your interests over time.

To minimize these threats, you can turn your default browser settings to exclude cookies especially third party cookies, since they can be used to build up detailed profiles of your surfing patterns over time.

##### 3. Browse Privately wherever possible

To avoid the tracking by websites, you should try to browse Internet privately wherever possible. This way websites would not be able to store cookies on your computer, which give information about your search pattern and surf history.

##### 4. Be careful while posting on Internet

You should know that posting is public. When you post anything to a public Internet such as *social networking sites* like Instagram or Facebook etc., *newsgroup*, *mailing list*, or *chat room*, you generally give up the rights to the content and any expectation of privacy or confidentiality. In

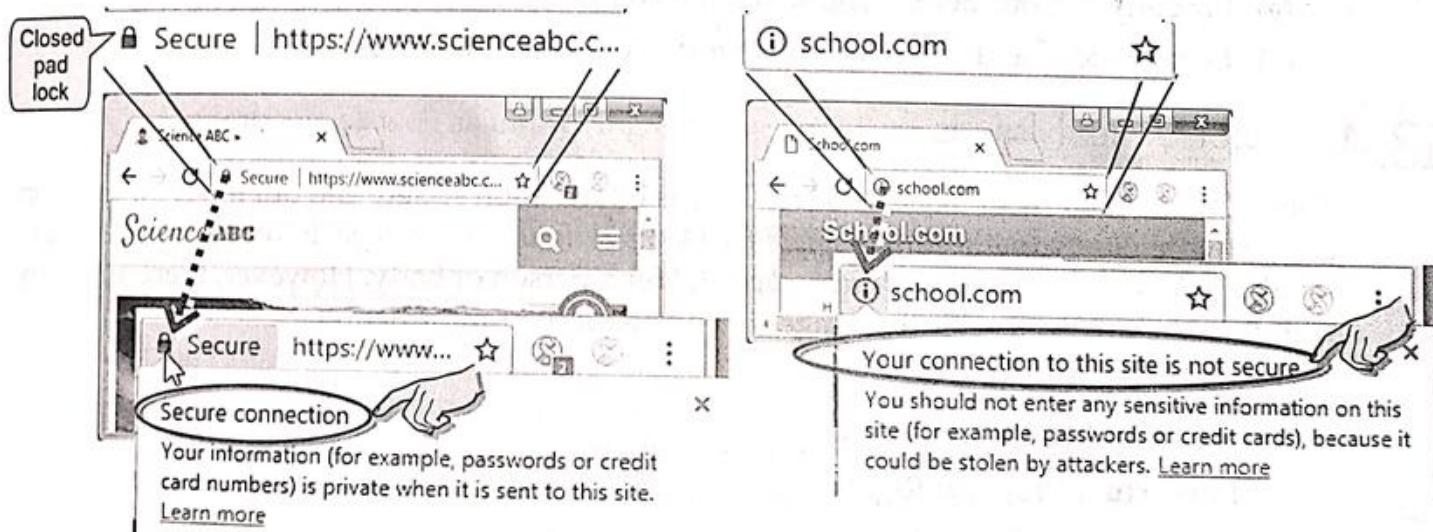
most countries, anything you post to a public space can be saved, archived, duplicated, distributed, and published, even years later, by anyone in the same way as a photograph taken in a public space like a city park.

So ensure that you never post your crucial information like your personal details like address, mobile phone number, bank details, credit card details etc. on public Internet sites.

### 5. Ensure Safe sites while entering crucial information

Sometimes, you have a need to provide your crucial information such as your personal details or bank details etc. For example, you might be applying online to register for an entrance exam through a legitimate site that asks for your personal details. In such case, ensure these things :

- ❖ Type the URL of the website in the address bar of the browser on your own . Do not click on a link that takes to this website; or do not cut/copy the link of this website and paste it. **TYPE THE URL ON YOUR OWN** in the address bar of the web browser.
- ❖ **Ensure that the address contains HTTP and a pad lock sign.** When this website gets loaded, before you start typing any information, ensure that the website address. A safe site's URL starts with **https://** and not with **http://**. Also, it shows a closed pad lock.



**https://** in a URL means it is a **secure connection** and no one can see your private information.

When there is no https or the URL contains only http, then it is an **insecure connection**, which means your private information may get leaked.

### 6. Carefully Handle Emails

While opening an email, make sure that you know the sender. Even if open the email message by accident, make sure not to open attachment in an email from unrecognized source. Emails containing sensitive information should be deleted securely.

Also, your email might contain a link to legitimate looking website; never click on any link inside an email to open it. The link might look legit but it may take you to a fraudulent site. Even if you need to visit the linked website, type the URL of the website on your own in the address bar of a web browser but never open any link inside an email.

### 7. Do not give sensitive information on wireless networks

Sometimes, you get access to some wireless connections such as the Wi-Fi connections available on Airports or Railway stations. While using such Wi-Fi connections, make sure not to open any sensitive information or provide any sensitive information on a website. The reason for this is that most free wireless networks are not encrypted and hence information on it can be tapped and used for fraudulent purposes.

### 8. Avoid using public computers

Always try not to use the public computer especially if have to deal with your crucial data. But if you need to work on a public computer, then make sure these things :

- (a) Browse privately, first of all.
- (b) Don't save your login information.
- (c) Never save passwords while working on a public computer.
- (d) Avoid entering sensitive information onto a public computer.
- (e) Don't leave the computer unattended with sensitive information on the screen.
- (f) Disable the feature that stores passwords.
- (g) Properly log out before you leave the computer.
- (h) Erase history and traces of your work, i.e., clear history and cookies.

## 13.4 Major Ethical Issues

These days, we can easily say that our society is information society and our era is information era. As we all know that *information is the means to acquire knowledge*. In other words, we can say that *information forms the intellectual capital* for a person or body. However, there are many ethical issues involved with the usage and availability of information.

These major ethical issues are :

1. Individual's Right to Privacy
2. Intellectual Property Rights
3. Accuracy of Information

### 13.4.1 Individual's Right to Privacy

Individual's right to privacy, in context of computing and information, pertains to following three issues :

1. Collecting information
2. Storing information
3. Distributing information

The right to privacy also involves the decisions pertaining to questions like *What information about one's self or one's associations must a person reveal to others, under what conditions and with what safeguards? What things can people keep to them and not be forced to reveal to others?*

With the growth of information technology and with its enhanced capacity for surveillance, communication, computation, storage, and retrieval, it can easily be used to invade one's privacy. Also with the increased value of information in decision-making, this threat becomes more serious.

Ethical issues pertaining to this, involve that one must not use computers to collect, store or distribute information that is owned by someone else. That is, one must not invade one's privacy.

#### 13.4.2 Intellectual Property Rights

As mentioned earlier, information makes intellectual property. Any piece of information is produced or created with a lot of efforts and it consumes a lot of time. The cost factor is also involved with the creation or production of information. Though once produced, it becomes very easy to duplicate it or share it with others. But this very thing makes information difficult to safeguard unlike tangible property.

The creator/producer of the information is the real owner of the information. And the owner has every right to protect his/her intellectual property. To protect one's intellectual property rights one can get information copyrighted or patented or use trademarks.

The ethical issue involved with it is that information must not be exchanged without the consent of its owner. The intellectual property rights must be protected for it :

- ❖ encourages individuals and businesses to create new software and new software applications, as well as improving existing applications,
- ❖ ensures new ideas and technologies are widely distributed,
- ❖ promotes investment in the national economy.

#### Note

*Intellectual property rights* are the rights of the owner of information to decide how much information is to be exchanged, shared or distributed. Also it gives the owner a right to decide the price for doing (exchanging/sharing/distributing) so.

#### 13.4.3 Accuracy of Information

With the value and importance of information, it becomes very much necessary to ensure that the information is accurate. When a user pays for some information or software, he/she has every right to ensure that the information or software being given to him/her is authentic, authorized and accurate. The ethical issue involved here is that the information owner should make available the authentic, authorized and accurate information to its purchaser.

Even if a user has received authentic, authorized and accurate information, there are many threats to its protection for there are many users that work with malicious intent and try to harm others by destroying important information. After buying a legal software/information, it becomes responsibility of the user to ensure its security. We shall cover data protection measures in a separate section.

### 13.5 Open Source and Its Philosophy

Broadly the term '*open source software*' is used to refer to those categories of software / programs whose licenses do not impose much conditions. Such software, generally, give users freedom to run/use the software for any purpose, to study and modify the program, and to redistribute copies of either the original or modified program (without having to pay royalties to previous developers).

There are many categories of software that may be referred to as open source software. Following subsection is going to talk about the same.

### 13.5.1 Terminology

Before we talk about various terms and definitions pertaining to 'Open' world, you must be clear about two terms which are often misunderstood or misinterpreted.

These terms are :

- ◆ Free software and
- ◆ Open source software

#### Free Software

*Free Software* means the software is freely accessible and can be freely used, changed, improved, copied and distributed by all who wish to do so. And no payments are needed to be made for **free software**.

The definition of **Free Software** is published by *Richard Stallman's Free Software Foundation*. Here is the key text<sup>1</sup> of that definition :

"*Free software*" is a matter of liberty, not price. To understand the concept, you should think of "free" as in "free speech," not as in "free beer." *Free software* is a matter of the users' freedom to run, copy, distribute, study, change and improve the software. More precisely, it refers to four kinds of freedom, for the users of the software :

- ◆ The freedom to run the program, for any purpose (freedom 0).
- ◆ The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
- ◆ The freedom to redistribute copies so you can help your neighbor (freedom 2).
- ◆ The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

A program is free software if users have all of these freedoms.

#### Open Source Software

*Open Source Software*, on the other hand, can be freely used (in terms of making modifications, constructing business models around the software and so on) but it *does not have to be free of charge*. Here the company constructing the business models around *open source software* may receive payments concerning support, further development. What is important to know here is that in *open source software*, the source code is freely available to the customer.

### 13.5.2 Philosophy of Open Source

Open source software is officially defined by the **open source definition** at [http://www.opensource.org/docs/definition\\_plain.html](http://www.opensource.org/docs/definition_plain.html). It states that :

Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria :

|                            |                                                                                                                                                             |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Free Redistribution</i> | No restriction on the re-distribution of the software whether as a whole or in part.                                                                        |
| <i>Source Code</i>         | The program must include source code, and must allow distribution in source code as well as compiled form.                                                  |
| <i>Derived Works</i>       | The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software. |

1. Excerpt courtesy *Free Software Foundation*. This keytext is available at [www.gnu.org/philosophy/free-sw.html](http://www.gnu.org/philosophy/free-sw.html).

|                                                                                                    |                                                                                                                                                                                                                                               |
|----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Integrity of the Author's Source Code                                                              | The integrity of the author/source code must be maintained. Any additions / modifications should carry a different name or version number from the original software.                                                                         |
| No Discrimination Against Persons or Groups                                                        | The license must not discriminate against any person or group of persons.                                                                                                                                                                     |
| No Discrimination Against Fields of Endeavor                                                       | The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.                     |
| Distribution of License                                                                            | The rights attached to the program must apply to all to whom the program is redistributed.                                                                                                                                                    |
| License must not be Specific to a Product                                                          | There must not be any restriction on the rights attached to the program, i.e., there should not be a condition on the program's being part of a particular software distribution.                                                             |
| The License must not Restrict other Software                                                       | The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software. |
| License must be Technology Neutral                                                                 | No provision of the license may be predicated on any individual technology or style of interface.                                                                                                                                             |
| A software which is free as well as open belongs to category FOSS (Free and Open Source Software). | The terms Free and Open represent a differing emphasis on importance of <b>freedom</b> (free software) or <b>technical progress</b> (open source software).                                                                                   |

### 13.5.3 Definitions

After understanding the difference between the terms free and open, let us now proceed to our discussion on terminology and definitions pertaining to open source software.

|               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| OSS and FLOSS | OSS refers to <i>open source software</i> , which refers to software whose source code is available to customers and it can be modified and redistributed without any limitation. An OSS may come free of cost or with a payment of nominal charges that its developers may charge in the name of development, support of software.<br>FLOSS refers to <i>Free Libre and Open Source Software</i> or to <i>Free Livre and Open Source Software</i> . The term FLOSS is used to refer to a software which is both <i>free software</i> as well as <i>open source software</i> . Here the words <i>libre</i> (a Spanish word) and <i>livre</i> (a Portuguese word) mean <i>freedom</i> . |
| GNU           | GNU <sup>2</sup> refers to <i>GNU's Not Unix</i> . GNU Project emphasizes on freedom. The GNU project was initiated by Richard M. Stallman with an objective to create an operating system. With time, GNU project expanded and now it is not limited to only an operating system. Now, it offers a wide range of software, including applications apart from operating system.                                                                                                                                                                                                                                                                                                        |
| FSF           | FSF is <i>Free Software Foundation</i> . FSF is a non-profit organization created for the purpose of supporting free software movement. Richard Stallman founded FSF in 1985 to support GNU project and GNU licences. Now a days, it also works on legal and structural issues for the free software community.                                                                                                                                                                                                                                                                                                                                                                        |

2. GNU is recursive acronym for GNU's Not Unix. A recursive acronym is the one that uses its abbreviation in full form e.g., VISA is also recursive acronym – VISA International Service Association.

**OSI**

OSI is *Open Source Initiative*. It is an organization dedicated to cause of promoting open source software. *Bruce Perens* and *Eric Raymond* were the founders of OSI, that was founded in February 1998.

OSI specifies the criteria for open source software and properly defines the terms and specifications of *open source software*.

Open source doesn't just mean access to the source code. The distribution terms of open source software must comply with the *Open Source Definition* by OSI.

**Freeware**

The term freeware is generally used for software, which is available free of cost and which allows copying and further distribution, but not modification and whose source code is not available. Freeware should not be mistaken for open software or for free software. Freeware is distributed in binary form (ready to run) without any licensing fee. In some instances the right to use the software is limited to certain types of users, for instance, for private and non-commercial purposes. One example is Microsoft Internet Explorer, which is made available as freeware.

**W3C**

W3C is acronym for *World Wide Web Consortium*. W3C is responsible for producing the software standards for world wide web. The W3C was created in October 1994, to lead the world wide web to its full potential by developing common protocols that promote its evolution and ensure its interoperability.

The World Wide Web Consortium (W3C) describes itself as follows :

The World Wide Web Consortium exists to realize the full potential of the Web.

The W3C is an industry consortium that seeks to promote standards for the evolution of the Web and interoperability between WWW products by producing specifications and reference software. Although industrial members fund W3C, it is vendor-neutral, and its products are freely available to all.

**Proprietary Software**

**Proprietary software** is the software that is *neither open nor freely available*. Its use is regulated and further distribution and modification is either forbidden or requires special permission by the supplier or vendor. Source code of proprietary software is normally not available.

**Shareware**

**Shareware** is software, which is made available with the right to redistribute copies, but it is stipulated that if one intends to use the software, often after a certain period of time, then a license fee should be paid.

Shareware is not the same thing as *free and open source software* (FOSS) for two main reasons : (i) the source code is not available and, (ii) modifications to the software are not allowed.

The objective of shareware is to make the software available to try for as many users as possible. This is done in order to increase prospective users' will to pay for the software. The software is distributed in binary form and often includes a built-in timed mechanism, which usually limits functionality after a trial period of usually one to three months.

**Copylefted Software**

**Copylefted software** is free software whose distribution terms ensure that all copies of all versions carry more or less the same distribution terms. This means, for instance, that copyleft licenses generally disallow others to add additional requirements to the software and require making source code available. This shields the program, and its modified versions, from some of the common ways of making a program proprietary.

### 13.5.4 Licenses and Domains of Open Source Technology

As per Open Source Initiative, "Open source licenses are licenses that comply with the Open Source Definition – in brief, they allow software to be freely used, modified, and shared."

Open-source licenses make it easy for others to contribute to a project without having to seek special permission. It also protects you as the original creator, making sure you at least get some credit for your contributions. It also helps to prevent others from claiming your work as their own. Broadly used open source licences are being given below for your reference :

#### 1. GNU General Public License (GPL)

The GNU General Public Licence (GPL) is probably one of the most commonly used licenses for open-source projects. The GPL grants and guarantees a wide range of rights to developers who work on open-source projects. Basically, it allows users to legally copy, distribute and modify software. This means, with GPL, a user can :

|                                                             |                                                                                                                                                           |
|-------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Copy the software</i>                                    | Copy the software as many times as needed. There's no limit to the number of copies one can make.                                                         |
| <i>Distribute the software however you want</i>             | There is no restriction of distribution methods and styles – can be in copied form or printed form or web-link form.                                      |
| <i>Charge a fee to distribute the software</i>              | After modifying the software, you can even charge for your software, explaining why you are charging them but the software should still be under GNU GPL. |
| <i>Make whatever modifications to the software you want</i> | You are free to make any kind of modifications to the GNU GPL software. The only catch is that the other project must also be released under the GPL.     |

#### 2. GNU Lesser General Public License (LGPL)

There is another GNU license : the Lesser General Public Licence(LGPL). It offers lesser rights to a work than the standard GPL licence. The LGPL is used to license *free software* so that it can be incorporated into both *free software* and *proprietary software*. The LGPL and GPL licenses differ with one major exception ; with LGPL the requirement that you have to release software extensions in open GPL has been removed.

Mostly, LGPL is used by libraries. LGPL is also called GNU libraries and formally called the Library GPL.

#### 3. BSD License

BSD licenses represent a family of permissive free software licenses that have fewer restrictions on distribution compared to other free software licenses such as the *GNU General Public License*. There are two important versions of BSD licence :

##### *the New BSD License/Modified BSD License*

The New BSD License ("3-clause license") allows unlimited redistribution for any purpose as long as its copyright notices and the license's disclaimers of warranty are maintained. The license also contains a clause restricting use of the names of contributors for endorsement of a derived work without specific permission.

##### *the Simplified BSD License / FreeBSD License*

The Simplified BSD license is different from New BSD License in the sense that the latter omits the non-endorsement clause.

#### 4. MIT License

The MIT License is the shortest and probably broadest of all the popular open-source licenses. Its terms are very loose and more permissive than most other licenses. The basic provisions of the license are :

- ❖ You can use, copy and modify the software however you want. No one can prevent you from using it on any project, from copying it however many times you want and in whatever format you like, or from changing it however you want.
- ❖ You can give the software away for free or sell it. You have no restrictions on how to distribute it.
- ❖ The only restriction is that it be accompanied by the license agreement. It basically says that anyone can do whatever they want with the licensed material, as long as it's accompanied by the license.

**Note** The MIT License is the least restrictive open source license.

#### 5. Apache License

The Apache License, grants a number of rights to users. These rights can be applied to both *copyrights* and *patents*. The Apache License offers :

*Rights are perpetual* Once granted, you can continue to use them forever.

*Rights are worldwide* If the rights are granted in one country, then they're granted in all countries.

*Rights are granted for no fee or royalty.* There is up-front usage fee, no per-usage fee or any other basis either.

*Rights are non-exclusive.* You are not the sole-licensee; other can also use the licensed work.

*Rights are irrevocable* No one can take these rights away once they're granted.

Redistributing code requires giving proper credit to contributors to the code and the same license (Apache) would remain with the software extension..

#### Public Domain Software vs. Proprietary Software

*Public-domain software* is free and can be used without restrictions. The term public-domain software is often used incorrectly to include freeware, free software that is nevertheless copyrighted. *Public domain software* is, by its very nature, outside the scope of copyright and licensing.

On the contrary, there is *Proprietary software*, which is neither free nor available for public. There is a proper license attached to it. User has to buy the licence in order to use it.

Consider the diagram (Fig. 13.3) originally made by Chao-Kuei<sup>3</sup> that describes the categories of software.

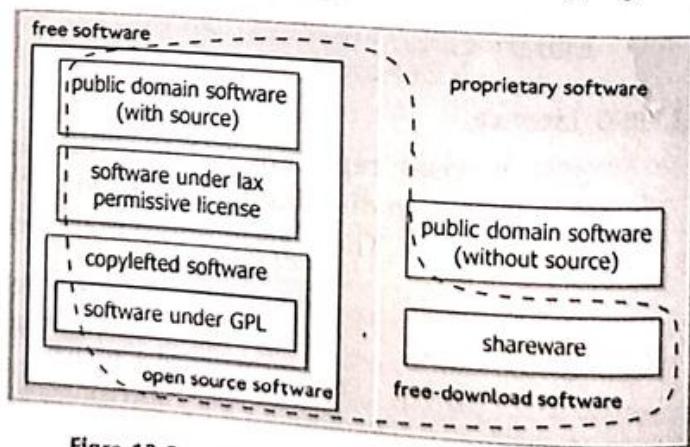


Fig 13.3 Categories and domains of software

3. and available under GNU GPL v2.

## 13.6 Netiquette

The word **netiquette**, derives from the combination of words - 'net' (internet) and 'etiquette'. It refers to online manners while using Internet or working online. While online, you should be courteous, truthful and respectful of others.

Following lines list basics rules of netiquettes :

1. **Refrain from personal abuse.** If you disagree to something, you may express robust disagreement, but never call them names or threaten them with personal violence.
2. **Never spam.** That is, don't repeatedly post the same advertisement for products or services.
3. **Write clearly and concisely.** On a site that has many non-native English speakers, avoid using slang they may not understand.
4. **Always post correct content in respectful language.** Remember that your posts are public. They can be read by your friends, your siblings, your parents, or your teachers.
5. **In a discussion forum, stick to the topic.** Don't post about football in a hair-care forum or about hair care in a gardening forum!
6. **Never expect other people to do your homework for you.** Never ask for or expect that your complete homework solution will be made available to you. Also, while asking for help, include details of what attempts you've made to solve the problem. It will save time and also show people that you are making an effort to help yourself.
7. **Do not post copyrighted material to which you do not own the rights.** It is plagiarism. If you need to use the copyrighted material, then follow these rules :
  - (a) Ask permission from the copyright holder.
  - (b) Enclose in quotes the copyrighted information that you are using and cite its source.

Failing this will put you and the site in legal trouble.

## 13.7 Email Etiquette

Though electronic mail has become a very popular way of communication yet most people using it are unaware of the etiquettes required for sending, receiving, using e-mails. This section is going to explain different e-mail etiquettes and the way to enforce them.

### What are the etiquette rules ?

There are many etiquette guides and many different etiquette rules. Some rules will differ according to the nature of your work, need and requirements. In the following lines, we are giving some most important email etiquette rules that apply to nearly all type of environments. These are :

1. **Be concise and to the point.** Do not make an e-mail longer than it needs to be. Remember that reading an e-mail is harder than reading printed communications and a long e-mail can be very discouraging to read.
2. **Use proper spelling, grammar & punctuation.** This is not only important because improper spelling, grammar and punctuation give a bad impression, it is also important for

conveying the message properly. E-mails with no full stops or commas are difficult to read and can sometimes even change the meaning of the text. And, if your program has a spell checking option, why not use it?

**3. Make it personal.** Not only should the e-mail be personally addressed, it should also include personal i.e., customized content. For this reason auto replies are usually not very effective.

**4. Answer swiftly.** E-mails are generally sent for fast responses. Therefore, each e-mail should be replied to within at least 24 hours, and preferably within the same day.

**5. Do not attach unnecessary files.** Only necessary files should be attached to mail. Wherever possible, try to compress attachments and ensure that the file being attached is virus free.

**6. Use proper structure and layout.** Since reading from a screen is more difficult than reading from paper, the structure and layout is very important for e-mail messages. Use short paragraphs and blank lines between each paragraph. When making points, number them or mark each point as separate to keep the overview.

**7. Do not write in CAPITALS.** IF YOU WRITE IN CAPITALS IT SEEMS AS IF YOU ARE SHOUTING. This can be highly annoying and might trigger an unwanted response in the form of a flame mail. Therefore, try not to send any email text in capitals.

**8. Read the email before you send it.** A lot of people don't bother to read an email before they send it out, as can be seen from the many spelling and grammar mistakes contained in emails. Apart from this, reading your email through the eyes of the recipient will help you send a more effective message and avoid misunderstandings and inappropriate comments.

**9. Do not overuse Reply to All.** Only use Reply to All if you really need your message to be seen by each person who received the original message.

**10. Mailings > use the Bcc: field or do a mail merge.** When sending an email mailing, some people place all the email addresses in the To : field.

There are two drawbacks to this practice:

(a) the recipient knows that you have sent the same message to a large number of recipients, and

(b) you are publicizing someone else's email address without their permission.

One way to get round this is to place all addresses in the Bcc : field. With this recipient will only see the address from the To : field in their email.

**11. Take care with abbreviations and emoticons.** In business emails, try not to use abbreviations and emoticons (such as BTW or : - ). The recipient might not be aware of the meanings of the abbreviations and emoticons, thus, it is better not to use it.

**12. Be careful with formatting.** Remember that when you use formatting in your emails, the sender might not be able to view formatting, or might see different fonts than you had intended. When using colors, use a color that is easy to read on the background.

**13. Do not forward chain letters.** Do not forward chain letters. We can safely say that all of them are hoaxes. Just delete the letters as soon as you receive them.

**14. Do not copy a message or attachment without permission.** Do not copy a message or attachment belonging to another user without permission of the originator. If you do not ask permission first, you might be infringing on copyright laws.

**15. Do not use email to discuss confidential information.** Sending an email is like sending a postcard. If you don't want your email to be displayed on a bulletin board, don't send it. Moreover, never make any libelous, sexist or racially discriminating comments in emails, even if they are meant to be a joke.

**16. Use a meaningful subject.** Try to use a subject that is meaningful to the recipient as well as yourself.

**17. Don't send or forward emails containing libelous, defamatory, offensive, racist or obscene remarks.** By sending or even just forwarding one libelous, or offensive remark in an email, you can face court cases resulting in penalties.

## L et Us Revise

- ❖ Intellectual property rights are the rights of the owner of information to decide how much information is to be exchanged, shared or distributed.
- ❖ Spamming refers to the sending of bulk mail by an identified or unidentified source.
- ❖ Artificial intelligence (AI) refers to the ability of a machine or a computer program to think, learn and evolve.
- ❖ The IoT (Internet of Things) is a new age technology that allows computing devices to transfer data over a network like Internet without requiring human-to-human or human-to-computer interaction.
- ❖ Virtual Reality(VR) is a technology that allows people to experience and interact in a 3D virtual environment that appears and feels like a real environment with the use of an electronic equipment.
- ❖ The Augmented Reality (AR) is a technology that transforms the view of physical real-world environment with superimposed computer-generated images, thus changing the perception of reality.
- ❖ A system designed to prevent unauthorized access to or from a private network is called firewall.
- ❖ Always ensure to maintain your online privacy.
- ❖ Be correct, truthful and respectful while online or while sending emails.

## S olved Problems

### 1. What is IoT ?

**Solution.** IoT stands for Internet of Things. It is basically a network using which things can communicate with each other using internet as means of communication between them. All the things should be IP protocol enabled in order to have this concept possible. Not one but multiple technologies are involved to make IoT a great success.

### 2. What is Augmented Reality ?

**Solution.** Augmented reality (AR) is a live direct or composite view of a physical, real-world environment superimposed with virtual elements, which have augmented (enhanced) by computer-generated sensory input such as sound, video, graphics or GPS data.

Most popular recent example of augmented reality is *Pokemon Go* game app.

**3. How is Augmented Reality different from Virtual Reality ?**

**Solution.** The Augmented Reality enhances the real world view of our world with superimposed technology generated imagery and information. What we view is a mix of real physical world plus digitally generated superimposed imagery/information. We can interact with what we view.

The virtual reality replaces our real world with a different virtual world which is digitally generated that feels like real world and we can interactively participate in it.

**4. What are individual's right to privacy ?**

**Solution.** The right to privacy also involves the decisions pertaining to questions like *What information about one's self or one's associations must a person reveal to others, under what conditions and with what safeguards? What things can people keep to them and not be forced to reveal to others?*

**5. What are intellectual property rights ?**

**Solution.** Intellectual property rights are the rights of the owner of information to decide how much information is to be exchanged, shared or distributed. Also it gives the owner a right to decide the price for doing (exchanging/sharing/distributing) so.

**6. Why should intellectual property rights be protected ?**

**Solution.** The intellectual property rights must be protected because protecting them

- ☒ encourages individuals and businesses to create new software and new software applications, as well as improving existing applications,
- ☒ ensures new ideas and technologies are widely distributed,
- ☒ promotes investment in the national economy.

**7. What do you understand by firewall ?**

**Ans.** The system designed to prevent unauthorized access to or from a private network is called firewall.

**8. How is firewall useful in ensuring network security ?**

**Ans.** A firewall is a network security system, either hardware- or software-based, that controls incoming and outgoing traffic based on a set of rules. A firewall grants or rejects network-traffic flow between Internet and a private or corporate network and thereby it ensures network security.

**9. What is meant by "Denial of Service" with reference to Internet service ?**

Or

*How does 'Denial of Service' attack disrupt an organization's network resources ?*

**Ans.** Denial-of-service (DoS) attacks are those attacks that prevent the legitimate users of the system, from accessing or using the resources, information, or capabilities of the system.

**10. Name two threats to security in a network. What is the role of Firewall in Network security ?**

**Ans.** Two threats to network security are :

(i) Snooping. Unauthorized access of someone else's data, e-mail, etc.

(ii) Eavesdropping. Secretly listening someone else private communication.

Firewall (both in hardware form or software form) prevents unauthorized access to or from a private network.

**11. List 2 measures to secure a network.**

**Ans.** Two measures to secure a network :

(i) Firewall. It helps to prevent unauthorized access to or from a private network.

(ii) Intrusion detection. It is the art and science of sensing when a system or network is being used inappropriately or without authorization.

## Glossary

**Augmented Reality (AR)** Technologically generated a real view which is mix of real physical world along with superimposed digitally generated imagery and information.

**Internet of Things** Technology that allows computing devices to transfer data over a network like Internet without requiring human-to-human or human-to-computer interaction.

**IoT** Internet of Things.

**Spam** Sending of bulk mail by an identified or unidentified source.

**Virtual Reality (VR)** Technologically generated view which is completely virtual but appears completely real.

## Assignments

1. What do you understand by Artificial Intelligence ?
2. What are new things that AI has made possible ?
3. What is Internet of Things ? What is its utility ?
4. What is Virtual Reality (VR) ?
5. List some uses of VR.
6. Enlist some VR devices.
7. What is Augmented Reality (AR) ?
8. How is Augmented Reality different from Virtual Reality ?
9. What measures would you take to ensure cyber safety and security ?
10. What is Phishing ? How can you safeguard yourself from it ?
11. What is firewall ?
12. What are open source based software ?
13. Compare and Contrast (i) Free software and Open source software (ii) OSS and FLOSS (iii) Proprietary software and Free software (iv) Freeware and Shareware (v) Freeware and Free software.
14. How is GPL different from BSD licences ?
15. What is netiquette ?
16. What are email etiquettes ?
17. What are major ethical issues in the field of computing ?
18. What do you understand by intellectual property rights ?
19. What is spamming ?
20. What is denial of service attack ?

## A P P E N D I X A

### Insertion Sort

Insertion sort is a popular sorting technique useful for sorting small arrays. The insertion sort technique is being explained below :

Suppose an array  $A$  with  $n$  elements  $A[1], A[2], \dots, A[N]$  is in memory. The insertion sort algorithm scans  $A$  from  $A[1]$  to  $A[N]$ , inserting each element  $A[K]$  into its proper position in the previously sorted subarray  $A[1], A[2], \dots, A[K - 1]$ .

That is :

**Pass 1.**  $A[1]$  by itself is trivially sorted.

**Pass 2.**  $A[2]$  is inserted either before or after  $A[1]$  so that :  $A[1], A[2]$  is sorted.

**Pass 3.**  $A[3]$  is inserted into its proper place in  $A[1], A[2]$ , that is, before  $A[1]$ , between  $A[1]$  and  $A[2]$ , or after  $A[2]$ , so that :  $A[1], A[2], A[3]$  is sorted.

**Pass 4.**  $A[4]$  is inserted into its proper place in  $A[1], A[2], A[3]$  so that :

$A[1], A[2], A[3], A[4]$  is sorted.

**Pass N.**  $A[N]$  is inserted into its proper place in  $A[1], A[2], \dots, A[N-1]$  so that :

$A[1], A[2], \dots, A[N]$  is sorted.

This sorting algorithm is frequently used when  $n$  is small. For example, this algorithm is very popular with bridge players when they are first sorting their cards.

There remains only the problem of deciding how to insert  $A[K]$  in its proper place in the sorted subarray  $A[1], A[2], \dots, A[K-1]$ . This can be accomplished by comparing  $A[K]$  with  $A[K-1]$ , comparing  $A[K]$  with  $A[K-2]$ , comparing  $A[K]$  with  $A[K-3]$ , and so on, until first meeting an element  $A[J]$  such that  $A[J] < A[K]$ . Then each of the elements  $A[K-1], A[K-2], \dots, A[J + 1]$  is moved forward one location, and  $A[K]$  is then inserted in the  $J + 1$ st position in the array.

The algorithm is simplified if there always is an element  $A[J]$  such that  $A[J] < A[K]$ ; otherwise we must constantly check to see if we are comparing  $A[K]$  with  $A[1]$ . This condition can be accomplished by introducing a sentinel element  $A[0] = -\infty$  (or a very small number).

#### Example

Suppose an array  $A$  contains 8 elements as follows :

77, 33, 44, 11, 88, 22, 66, 55

Sort this array using Insertion Sort.

**Solution.** Table illustrates the insertion sort algorithm. The circled element indicates the  $A[K]$  in each pass of the algorithm, and the arrow indicates the proper place for inserting  $A[K]$ .

## Insertion sort for n = 8 items

| Pass     | A[0] | A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |
|----------|------|------|------|------|------|------|------|------|------|
| K = 1:   | - ∞  | 77   | 33   | 44   | 11   | 88   | 22   | 66   | 55   |
| K = 2 :  | - ∞  | 77   | 33   | 44   | 11   | 88   | 22   | 66   | 55   |
| K = 3 :  | - ∞  | 33   | 77   | 44   | 11   | 88   | 22   | 66   | 55   |
| K = 4 :  | - ∞  | 33   | 44   | 77   | 11   | 88   | 22   | 66   | 55   |
| K = 5 :  | - ∞  | 11   | 33   | 44   | 77   | 88   | 22   | 66   | 55   |
| K = 6 :  | - ∞  | 11   | 33   | 44   | 77   | 88   | 22   | 66   | 55   |
| K = 7 :  | - ∞  | 11   | 22   | 33   | 44   | 77   | 88   | 22   | 55   |
| K = 8 :  | - ∞  | 11   | 22   | 33   | 44   | 66   | 77   | 88   | 55   |
| Sorted : | - ∞  | 11   | 22   | 33   | 44   | 55   | 66   | 77   | 88   |

The Java program to implement Insertion Sort is given below :

```

public class InsertSort {
 static int data[] = { 3, 2, 4, 5, 1, 10, 9, 7, 6, 8 }; // initial data
 static int temp; // value to be inserted
 static int i; // index of value to be inserted
 static int loc = 0; // index used to find location for insertion
 static void dataPrint() { // print array content
 System.out.print("run() executed: i = " + I + ", data = ");
 for (int i = 0; i < data.length; i++) {
 System.out.print(data[i]); // print each value in data[]
 if (i < data.length - 1) System.out.print(", ");
 }
 System.out.println("");
 }
 public static void main(String argv[]) {
 System.out.println("Sorting (re)started");
 i = 0; // i increases from 1 to data.length - 1
 dataPrint();
 for (i++; i < data.length; i++) { // perform sorting
 temp = data[i]; // move value to be inserted to temp
 loc = i; // loc decreases from i to 0
 for (; loc >= 0; loc--) {
 if (loc == 0 || data[loc-1] <= temp) {
 data[loc] = temp; // insert value in temp
 break;
 }
 else {
 data[loc] = data[loc-1]; // move current value down
 }
 }
 dataPrint();
 }
 System.out.println("Sorting completed");
 }
}

```

## Methods in Wrapper Class

### The Integer Class

The Integer class is wrapper class for the primitive data type int the Integer class wraps an int value in an object. The Integer class provides methods for converting an int to a String object and a String object to an int as shown in Table 1.

The main difference between the parseInt( ) and valueOf( ) methods is that first one returns an int value whereas later one returns Integer object.

**Table 1 Methods of Integer Class**

| Methods                          | Description                                   |
|----------------------------------|-----------------------------------------------|
| static int parseInt(String s)    | Converts a String s to an int using radix 10. |
| static Integer valueOf(String s) | Converts a String s to an Integer object.     |
| static String toString(int n)    | Converts an int n to a String object.         |
| String toString( )               | Converts an invoking object to a String.      |

### The Byte Class

The Byte class is wrapper class for the primitive datatype byte. The Byte class wraps a byte value in an object. The Byte class provides methods for converting a byte to a String object and a String object to a byte as shown in Table 2.

**Table 2 Methods of Byte Class**

| Methods                         | Description                              |
|---------------------------------|------------------------------------------|
| static byte parseByte(String s) | Converts a String s to a byte.           |
| static Byte valueOf(String s)   | Converts a String s to a Byte object.    |
| static String toString(byte n)  | Converts an byte n to a String object.   |
| String toString( )              | Converts an invoking object to a String. |

### The Short Class

The Short class is wrapper class for the primitive datatype short. The Short class wraps a short value in an object. The Short class provides methods for converting a short to a String object and a String object to a short as shown in Table 3.

**Table 3 Methods of Short Class**

| <b>Methods</b>                    | <b>Description</b>                       |
|-----------------------------------|------------------------------------------|
| static short parseShort(String s) | Converts a String s to a short.          |
| static Short valueOf(String s)    | Converts a String s to a Short object.   |
| static String toString(short n)   | Converts an short n to a String object.  |
| String toString( )                | Converts an invoking object to a String. |

### The Long Class

The Long class is wrapper class for the primitive datatype long. The Long class wraps a long value in an object. The Long class provides methods for converting a long to a String object and a String object to a long as shown in Table 4.

**Table 4 Methods of Long Class**

| <b>Methods</b>                 | <b>Description</b>                       |
|--------------------------------|------------------------------------------|
| static long parseInt(String s) | Converts a String s to a long.           |
| static Long valueOf(String s)  | Converts a String s to an Long object.   |
| static String toString(long n) | Converts an long n to a String object.   |
| String toString( )             | Converts an invoking object to a String. |

### The Float Class

The Float class is wrapper class for the primitive data type float. The Float class wraps a float value in an object. The Float class provides methods for converting a float to a String object and a String object to a float as shown in Table 5.

**Table 5 Methods of Float Class**

| <b>Methods</b>                    | <b>Description</b>                       |
|-----------------------------------|------------------------------------------|
| static float parseFloat(String s) | Converts a String s to a float.          |
| static Float valueOf(String s)    | Converts a String s to an Float object.  |
| static String toString(float n)   | Converts an float n to a String object.  |
| String toString( )                | Converts an invoking object to a String. |

### The Double Class

The Double class is wrapper class for the primitive datatype Double. The Double class wraps a double value in an object. The Double class provides methods for converting a double to a String object and a String object to a double as shown in Table 6.

**Table 6 Methods of Double Class**

| <b>Methods</b>                      | <b>Description</b>                       |
|-------------------------------------|------------------------------------------|
| static double parseDouble(String s) | Converts a String s to a double.         |
| static Double valueOf(String s)     | Converts a String s to an Double object. |
| static String toString(double n)    | Converts an double n to a String object. |
| String toString( )                  | Converts an invoking object to a String. |

### The Character Class

The Character class is a wrapper class for the primitive datatype and it wraps a value of the primitive datatype char in an object. The Character class provides a charValue( ) method that returns the char value contained in a Character object.

The Character class provides several methods, some important ones are shown in Table 7.

**Table 7 Methods of Character Class**

| <b>Methods</b>                          | <b>Description</b>                                                               |
|-----------------------------------------|----------------------------------------------------------------------------------|
| static boolean isDigit(char ch)         | Returns true if ch is a numeric digit between 0 abd 9 ; otherwise returns false. |
| static boolean isLetter(char ch)        | Returns true if ch is letter otherwise returns a false.                          |
| static boolean isLetterOrDigit(char ch) | Returns true if ch is a letter or a digit otherwise returns a false.             |
| static boolean isLowerCase(char ch)     | Returns true if ch is a lowercase otherwise returns a false.                     |
| static boolean isUpperCase(char ch)     | Returns true if ch is a uppercase otherwise returns a false.                     |
| static boolean isWhitespace(char ch)    | Returns true if ch is a whitespace otherwise returns a false.                    |
| static char toLowerCase(char ch)        | Returns a lowercase equivalent of ch.                                            |
| static char toUpperCase(char ch)        | Returns a uppercase equivalent of ch.                                            |

Some generally available methods on Wrapper classes are given below :

**Table 8 : General Methods**

| <b>Methods</b>        | <b>Description</b>                                    |
|-----------------------|-------------------------------------------------------|
| byte byteValue( )     | Returns the value of the invoking object as a byte.   |
| short shortValue( )   | Returns the value of the invoking object as a short.  |
| int intValue( )       | Returns the value of the invoking object as an int.   |
| long longValue( )     | Returns the value of the invoking object as a long.   |
| float floatValue( )   | Returns the value of the invoking object as a float.  |
| double doubleValue( ) | Returns the value of the invoking object as a double. |