



# Object-oriented Programming Using **C++** and **Java**

Ramesh Vasappanavara  
Anand Vasappanavara  
Gautam Vasappanavara



Object-oriented  
Programming Using  
**C++** and **Java**

Ramesh Vasappanavara  
Anand Vasappanavara  
Gautam Vasappanavara

ALWAYS LEARNING

PEARSON

# OBJECT-ORIENTED PROGRAMMING USING C++ AND JAVA

**Ramesh Vasappanavara**

*Director*

*Shree Ganpati Institute of Technology  
Ghaziabad, Delhi (NCR)*

**Anand Vasappanavara**

*Process Control Technologist*

*Shell Technologies  
Bintulu, Malaysia*

**Gautam Vasappanavara**

*Lead Engineer*

*Samsung India Ltd  
Bangalore*

**PEARSON**

Chennai • Delhi • Chandigarh

# Brief Contents

*About the Authors*

*Preface*

*How to Use This Book and Web Resources*

*1 Object-oriented Programming Basics*

*2 Object Modelling*

*3 Extensibility and Reusability – Inheritance at Work*

*4 Dynamic Modelling*

*5 Analysis and Design Methodologies*

*6 C++ Fundamentals and Basic Programming*

*7 C++ Programming Basics and Control Loops*

*8 Functions, Storage Class Preprocessor Directives, and  
Arrays and Strings*

*9 Pointers and References*

*10 Classes*

*11 C++ Special Features: Errors and Exceptions and  
Operator Overloading*

12 Inheritance

13 IO Streaming

14 Generic Programming and Templates

15 Object-oriented Programming with Java

16 Java Fundamentals and Control Loops

17 Simple IO and Arrays and Strings Vectors

18 Class Objects and Methods

19 Inheritance: Packages: Interfaces

20 Errors and Exceptions in Java and Multithreaded Programming

21 Java IO Files

22 Networking in Java

23 Graphics Using Swing Components and Applets

24 Collections and Software Development Using Java

*Appendix A*

*Appendix B*

*Appendix C*

# Contents

*About the Authors*

*Preface*

*How to Use This Book and Web Resources*

1 Object-oriented Programming Basics

1.1 Introduction

1.2 Programming Concepts

1.3 Programming Paradigms

1.3.1 Structured Programming Paradigm

1.3.2 C: A Workhorse that Works Well

1.3.3 Where is the Problem?

1.3.4 Object-oriented Programming Paradigm

1.4 History and Development of Object-oriented Languages

1.4.1 C++

1.4.2 Java

### 1.4.3 C#

## 1.5 Software Development Methodologies

## 1.6 Need for Objects

## 1.7 Object-oriented Language Features

### 1.7.1 Object-based Programming

### 1.7.2 Object-oriented Programming

## 1.8 Definition of OOP Language Classes and Objects

### 1.8.1 Attributes and Behaviours of Objects

### 1.8.2 Class

### 1.8.3 Encapsulation

### 1.8.4 Data Hiding/Data Abstraction

### 1.8.5 Function Overloading

### 1.8.6 Operator Overloading

## 1.9 Extendibility and Reusability of OOP Paradigms

## 1.10 Extending/Deriving New Classes

### 1.10.1 Containment: Class Within a Class – Container Class

### 1.10.2 Inheritance and Class Hierarchy

### 1.10.3 Single and Multiple Inheritances

## 1.11 Virtual Functions



## 1.12 Run-time Polymorphism and Dynamic Data Binding

### 1.13 Class as Abstract Data Type (ADT)

### 1.14 Standard Template Library (STL)

### 1.15 OOPS – Object-oriented Programming and Systems

### 1.16 Object-oriented Analysis and Design (OOAD)

### 1.17 Summary

#### *Exercise Questions*

#### *Objective Questions*

#### *Short-answer Questions*

#### *Long-answer Questions*

#### *Solutions to Objective Questions*

## 2 Object Modelling

### 2.1 Introduction

### 2.2 Object Model

### 2.3 Object-oriented Design

### 2.4 Classes and Objects

#### 2.4.1 Class

#### 2.4.2 Notations and Meta Models

#### 2.4.3 Mandatory Profile

2.4.4 Meta Data – Meta Class

2.4.5 Constraints

2.4.6 Object Creation

2.4.7 Garbage Collection

2.5 Object Properties

2.6 Links

2.7 Class Diagrams

2.8 Class Hierarchy

2.8.1 Associations

2.8.2 Hierarchies with Interdependent Classes

2.8.3 Hierarchies with Independent Classes

2.9 Object Diagrams

2.10 Communications and Message Passing

2.11 Polymorphism

2.12 Abstract Class

2.13 Concrete Class

2.13.1 OOPS – A Case Study in Object Modelling

2.14 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

### 3 Extensibility and Reusability – Inheritance at Work

#### 3.1 Extendibility and Reusability of OOP Paradigm

#### 3.2 Extending/Deriving New Classes

##### 3.2.1 Containment: Class within a Class – Container Class

##### 3.2.2 Inheritance and Class Hierarchy

##### 3.2.3 Generalization vs Specialization

#### 3.3 Overriding Base Class Functions

#### 3.4 Overloading

#### 3.5 Single and Multiple Inheritances

#### 3.6 Virtual Inheritance

#### 3.7 Problems with Multiple Inheritance

#### 3.8 Interface

#### 3.9 Access Specifiers in Inheritance

#### 3.10 Run-time-type Information – RTTI

#### 3.11 Virtual Functions

### 3.12 Pure Virtual Functions

### 3.13 Run-time Polymorphism and Dynamic Data Binding

### 3.14 Class as Abstract Data Type

### 3.15 Separation of Behaviour and Implementation

### 3.16 Decoupling

### 3.17 Standard Template Library

### 3.18 Summary

#### *Exercise Questions*

#### *Objective Questions*

#### *Short-answer Questions*

#### *Long-answer Questions*

#### *Assignment Questions*

#### *Solutions to Objective Questions*

## 4 Dynamic Modelling

### 4.1 Introduction to Static and Dynamic Modelling

### 4.2 Lifetime of an Object

### 4.3 Unified Modelling Language – A Tool for OOAD

#### 4.3.1 Overview of UML

#### 4.3.2 Classification of UML Diagrams

4.4 User Requirements – Use Cases

4.5 Architecture and Domain: What Are They?

4.5.1 Architecture

4.5.2 Domain

4.6 Sequence Diagram

4.7 Collaboration (Communication) Diagram

4.8 Events and State

4.8.1 Activity/Action and Operations

4.8.2 Nested State Diagrams

4.9 Activity Diagram

4.10 Packages and Components – A Way to Organize  
Large and Complex Projects

4.11 Component and Deployment Diagram

4.12 Summary

Exercise Questions

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

## 5 Analysis and Design Methodologies

### 5.1 Introduction

### 5.2 Stages and Methodologies for Systems Development

#### 5.2.1 Methodologies for Software Development

#### 5.2.2 Systems Development Life Cycle (SDLC)

### 5.3 Structured Analysis and Design (SASD)

#### 5.3.1 Conceptual Design – Functional Modelling and Data Modelling

#### 5.3.2 Modular Design

#### 5.3.3 Analysis and Design Techniques and Tools

#### 5.3.4 Context Diagrams

#### 5.3.5 Event List

#### 5.3.6 Data Flow Diagrams

#### 5.3.7 Data Dictionary

#### 5.3.8 Multilevel DFDs

#### 5.3.9 Entity Relationship Diagram

#### 5.3.10 Process Specifications

#### 5.3.11 Specifying Constraints

### 5.4 Case Study on Structured Analysis and Design

#### 5.4.1 Example of Statement of Purpose

#### 5.4.2 Context Diagram

#### 5.4.3 Event List

#### 5.4.4 Data Flow Diagrams

#### 5.4.5 Data Dictionary

#### 5.4.6 Entity Relationship Diagrams

#### 5.4.7 Process Specifications – SSCS

#### 5.4.8 Structured Analysis and Design – Advantages and Disadvantages

### 5.5 Object-oriented Software Analysis and Design (OOAD)

#### 5.5.1 Different Models for Object Analysis

#### 5.5.2 Identifying Classes

#### 5.5.3 Identifying Attributes

#### 5.5.4 Specifying Operations

#### 5.5.5 Work Out Associations

#### 5.6 OOAD: A Case Study

#### 5.7 Design for Reuse

#### 5.8 Comparison of SASD and OOAD Methodologies

#### 5.9 Summary

#### *Exercise Questions*

#### *Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

## 6 C++ Fundamentals and Basic Programming

### 6.1 Introduction

### 6.2 History and Development of Object-oriented Languages like C++

### 6.3 Object-oriented Language Features

### 6.4 Welcome to C++ Language

#### 6.4.1 Setting Path

#### 6.4.2 C++ Program Structure

#### 6.4.3 C++ Development Environment

### 6.5 Further Sample Programs of C++

#### 6.5.1 Execution of a Program in a Loop

#### 6.5.2 Arrays Implementation

#### 6.5.3 Use of Structure to Implement Problem

#### 6.5.4 Class Implementation

### 6.6 Console IO Operations



6.6.1 Console IO Functions – Commands `getchar()`  
and `putchar()`

6.6.2 Console IO Functions – Commands `gets` and  
`puts`

6.6.3 Unformatted Stream IO Functions – `get()`,  
`put()`, `getline()` and `write()`

## 6.7 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

## 7 C++ Programming Basics and Control Loops

7.1 Introduction

7.2 Declaration of Variables

7.3 Data Types

7.4 Declaration and Assignment Values to Variables

7.5 Expressions

7.6 Operators

7.6.1 IO Operators << and >>, iostream objects cin and cout

7.6.2 Unary Operators

7.6.3 Size of Operator

7.6.4 Arithmetic Operators

7.6.5 Relational and Logical Operators

7.6.6 Logical Operators

7.6.7 Conditional Expressions: Question Mark Operator

7.6.8 Comma Operator

7.6.9 Bitwise Operators

7.7 Precedence and Association of Operators

7.8 Control Loops

7.8.1 Conditional and Branching Statements

7.8.2 Switch and Case Statements

7.8.3 While Loop

7.8.4 Do-while Loop

7.8.5 For Loop

7.8.6 When to Use For or While or Do-while

7.9 Break and Continue

7.9.1 Break

7.9.2 Continue Statement

7.9.3 Goto Statements

7.9.4 Exit Function

7.10 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

8 Functions, Storage Class Preprocessor Directives, and  
Arrays and Strings

8.1 Introduction

8.2 Why Use Function Templates?

8.3 Call by Value

8.4 Call by Reference

8.5 Call by Constant Reference

8.6 Recursion

8.7 Inline Functions

## 8.8 Function Overloading

## 8.9 Default Arguments

## 8.10 Memory Management of C++

### 8.10.1 Types of Storage Classes

## 8.11 Header Files and Standard Libraries

## 8.12 C++ Preprocessor

### 8.12.1 Macro Expansion

### 8.12.2 Macro Definition with Arguments

### 8.12.3 File Inclusion

### 8.12.4 Conditional Inclusion

### 8.12.5 Conditional Compilation `#ifdef` and `#ifndef` Statements

### 8.12.6 `#undef`

### 8.12.7 `#error` Macros

## 8.13 Arrays

### 8.13.1 How are Arrays Stored in the Memory?

### 8.13.2 Array Initialization

### 8.13.3 Multi-dimensional Arrays

### 8.13.4 Character Array – String Handling in C++ Language

## 8.14 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

## 9 Pointers and References

9.1 Introduction

9.2 What, Why and How of Pointers

9.3 Pointers Declaration and Usage

9.4 Call by Value and Call by Reference (Pointers)

9.5 Dynamic Memory New and Delete Functions

9.5.1 Memory Leak

9.5.2 Dangling Pointer

9.5.3 Pointers and Arrays

9.5.4 Pointers and Two-dimensional Arrays

9.5.5 Array Declaration on Heap Memory

9.5.6 Pointer to Pointer

9.5.7 Dynamic Memory for a Two-dimensional Array

9.5.8 Pointers and Three-dimensional Arrays

9.5.9 Array of Pointers

9.5.10 Pointers to Void

9.5.11 Pointer to a Constant vs const Pointer

9.5.12 Pointers to Function

9.6 What, Why and How of References

9.6.1 Which is Better – Pointer or Reference?

9.7 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Solutions to Objective Questions*

10 Classes

10.1 Introduction

10.2 Classes and Objects

10.2.1 How to Create an Object to a Class?

10.2.2 How to Access Member Data and Member Functions?

10.2.3 Constructors and Destructors

10.3 Friend Function

10.4 Class Within a Class: Container Class

10.5 Objects and Data Members on Heap Memory

10.6 This Pointer

10.7 Pointers vs Objects: Use of Constant Declarations

10.8 Passing of Objects by Reference and Pointers to a Function

10.9 Constant Pointers and Constant References

10.10 Static Member Data

10.11 Static Member Functions

10.12 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Solutions to Objective Questions*

11 C++ Special Features: Errors and Exceptions and Operator Overloading

11.1 Introduction

11.2 Errors and Exceptions

11.3 Try and Catch Blocks

## 11.4 Exception Classes

## 11.5 Stack Data Structure

## 11.6 Operator Overloading

### 11.6.1 Overloading of + Operator and \* Operator

### 11.6.2 Operator << Overloading

### 11.6.3 What We Can Overload and What We Cannot Overload

### 11.6.4 Overloading Assignment Operator =

## 11.7 Pre- and Post-increment Operator Overloading

## 11.8 Overloading Operators New and Delete

## 11.9 Conversion (Casting) Operators

## 11.10 Summary

### *Exercise Questions*

#### *Objective Questions*

#### *Short-answer Questions*

#### *Long-answer Questions*

#### *Assignment Questions*

#### *Solutions to Objective Questions*

## 12 Inheritance

### 12.1 Introduction



12.2 Inheritance Hierarchy

12.3 Types of Inheritance

12.4 Constructors and Destructors

12.5 Base Class Function Overriding

12.6 Base Class Function Hiding

12.7 Virtual Functions

12.8 Multiple Virtual Functions

12.9 Virtual Destructors: Why and How?

12.10 Hybrid Inheritance with Multiple Inheritances

12.11 Virtual Inheritance

12.12 Run-time Polymorphism and Dynamic Binding

12.13 Abstract Data Types (ADTs)

12.14 Pure Virtual Functions

12.15 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

## 13 IO Streaming

### 13.1 Introduction

### 13.2 IO Streaming

### 13.3 IO Library Files Classification

#### 13.3.1 Formatted IO

#### 13.3.2 Unformatted IO

#### 13.3.3 IO Stream State

#### 13.3.4 IO Stream Library – Header Files

### 13.4 IO Manipulators

### 13.5 Flags

### 13.6 File I/O

#### 13.6.1 File Types

#### 13.6.2 Stream Operating Modes

### 13.7 Binary File

### 13.8 Seekg() / Seekp() and Tellg() and Tellp()

## Functionality of C++

### 13.9 Summary

#### *Exercise Questions*

#### *Objective Questions*

#### *Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

## 14 Generic Programming and Templates

### 14.1 Introduction

### 14.2 What is Generic Programming and Why Use Templates?

### 14.3 Template Classes

### 14.4 Function Templates and Passing of Arguments to a Function

### 14.5 Template Friends

#### 14.5.1 Non-template Function

#### 14.5.2 Template Friend Class or Function

#### 14.5.3 Type-specific Friend Function in Class Templates

### 14.6 Templates and Static Member Functions and Member Data

### 14.7 Template Exceptions

### 14.8 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

## 15 Object-oriented Programming with Java

### 15.1 Introduction

### 15.2 Internet and the World Wide Web

### 15.3 C and C++ are Around – Then Why Java?

### 15.4 Java Story

### 15.5 Java Features

#### 15.5.1 Portability or Platform Independent

#### 15.5.2 Automatic Garbage Collection

#### 15.5.3 Object-oriented Features

#### 15.5.4 Easy to Learn and Excellent Documentation

#### 15.5.5 Byte Code

#### 15.5.6 Java Virtual Machine (JVM)

#### 15.5.7 Comparison with C++

### 15.6 Developing First Java Application

#### 15.6.1 Installing and Using Java Development Kit

#### 15.6.2 Setting Path and Classpath

15.6.3 Java Program Structure

15.6.4 Java Documentation Comments

15.6.5 Java Development Environment

15.6.6 Our First Java Application

15.6.7 Application with Swing Components

15.6.8 Eclipse-integrated Development Environment

15.6.9 Command Line Arguments

15.7 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

16 Java Fundamentals and Control Loops

16.1 Introduction

16.2 Constants/Literal Constants

16.2.1 Integer Constants

16.2.2 Floating Point Constants/Real Constants

16.2.3 Character Constants

16.2.4 String Constants

16.2.5 Backlash Character Strings

16.2.6 Boolean Literals

16.2.7 Symbolic Constants

16.3 Variables and Assignment of Values to Variables

16.4 Data Types

16.4.1 Integer Data Types

16.4.2 Floating Point Data Types

16.4.3 Character Type

16.4.4 Boolean Data Type

16.5 Scope and Life Time of Variables

16.6 Arithmetic Operators

16.7 Type Conversion and Type Casting

16.7.1 Type Conversion

16.7.2 Type Cast

16.8 Unary Operators

16.8.1 Increment and Decrement Operators

16.8.2 Assignment Operator

16.8.3 Chained Assignment

16.8.4 Relational Operators

## 16.9 Logical Operators

## 16.10 Bit-wise Operators

## 16.11 Other Operators

### 16.11.1 Question Mark (?) Operator Conditional Expressions

### 16.11.2 Member Operator or Dot Operator

### 16.11.3 Instanceof Operator

### 16.11.4 New Operator

### 16.11.5 Operator Precedence and Associativity

## 16.12 Conditional and Branching Statements

### 16.12.1 If and If-Else Statements

### 16.12.2 Nested If Statements

### 16.12.3 If-Else-If Ladder

### 16.12.4 Switch and Case Statements

## 16.13 Control Loops

### 16.13.1 While Loop

### 16.13.2 Do-while Loop

### 16.13.3 For Loop

## 16.14 Break

## 16.15 Continue Statement

## 16.16 Summary

### *Exercise Questions*

#### *Objective Questions*

#### *Short-answer Questions*

#### *Long-answer Questions*

#### *Assignment Questions*

#### *Solutions to Objective Questions*

## 17 Simple IO and Arrays and Strings Vectors

### 17.1 Introduction

### 17.2 Input from Keyboard

#### 17.2.1 `System.in`, `System.out` and `System.err` Commands

#### 17.2.2 `StringTokenizer` to Receive Multiple Inputs in a Single Line

#### 17.2.3 Obtaining Inputs Using Java's Scanner Class

#### 17.2.4 Using Control Formats - `System.out.printf()`

#### 17.2.5 Formatted Output with String Format

### 17.3 Arrays

#### 17.3.1 Declaring and Creation of an Array

#### 17.3.2 Initialization of Arrays



17.3.3 How Are Arrays Stored in the Memory?

17.3.4 Accessing and Modifying Array Elements

17.3.5 Passing Arrays as Arguments to Methods

17.3.6 Returning Arrays as Arguments to Methods

17.3.7 Multi-dimensional Arrays

17.3.8 `Java.util. Arrays` Class

17.4 String

17.4.1 Array of Strings

17.4.2 String Class Methods

17.4.3 `StringBuffer` Class

17.4.4 `StringBuilder` Class

17.5 Collection Framework

17.5.1 Vector Class

17.5.2 Vector Methods

17.6 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

## *Assignment Questions*

## *Solutions to Objective Questions*

### 18 Class Objects and Methods

#### 18.1 Introduction

#### 18.2 Classes and Objects

#### 18.3 Declaring a Class and Creating of Instances of Class Variables

#### 18.4 Constructors

##### 18.4.1 Default Constructor

##### 18.4.2 Parameterized Constructor and Overloading of Constructors

#### 18.5 Specifying Private Access Specifiers and Use of Public Methods

##### 18.5.1 Private Access Specifiers

##### 18.5.2 Methods

##### 18.5.3 Math Class of Java

##### 18.5.4 Call by Value and Call by Reference

##### 18.5.5 Passing and Returning of Objects To and From Methods

##### 18.5.6 Method Overloading

#### 18.6 Usage of `this` Keyword

18.7 Garbage Collection

18.8 Finalizer and `Finalize()` Methods

18.9 Final Variable

18.10 Access Control and Accessing Class Members

18.11 Static Members

18.12 Factory Methods

18.13 Nested Classes

18.14 Inner Classes

18.15 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

19 Inheritance: Packages: Interfaces

19.1 Introduction

19.2 Basic Concepts of Inheritance

19.3 Member Access Rules

19.4 Using Super Class

19.5 Methods Overriding

19.6 Multilevel Inheritance

19.7 Run-time Polymorphism

19.8 Abstract Classes

19.9 Using Final with Inheritance

19.9.1 Final Method

19.9.2 Final Classes

19.10 Object Class

19.11 Packages

19.11.1 Reusable Classes

19.12 Path and Classpath

19.13 Importing of Packages

19.14 Access Specifiers Revisited for Packages

19.15 Interfaces

19.15.1 What and Why of Interfaces

19.15.2 Defining and Implementing Interfaces

19.16 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

## 20 Errors and Exceptions in Java and Multithreaded Programming

### 20.1 Introduction

### 20.2 Errors and Exceptions

#### 20.2.1 Errors

#### 20.2.2 Exceptions

### 20.3 Try and Catch Blocks

### 20.4 Handling of Multiple Exceptions by Try and Catch Blocks

### 20.5 Using Finally Block

### 20.6 Throw Exceptions

### 20.7 Throws Exceptions

### 20.8 Re-throwing of an Exception

### 20.9 Defining Our Own Exception

#### 20.9.1 Procedure for Throwing Our Own Exceptions

### 20.10 Concepts of Multithreading

#### 20.10.1 Multithreaded Program

### 20.11 Process vs Threads

### 20.11.1 Process

### 20.11.2 Threads

## 20.12 How to Create and Run the Threads?

### 20.12.1 Which is Better: Extends Thread or Implements Runnable?

### 20.12.2 Use of `isAlive()` and `join()` Methods

## 20.13 Life Cycle of Thread

## 20.14 Thread Priorities

## 20.15 Synchronization

## 20.16 Inter-thread Communications

## 20.17 Deadlock in Multithreaded Programming

## 20.18 Summary

### *Exercise Questions*

#### *Objective Questions*

#### *Short-answer Questions*

#### *Long-answer Questions*

#### *Assignment Questions*

#### *Solutions to Objective Questions*

## 21 Java IO Files

### 21.1 Introduction

## 21.2 IO Streaming

## 21.3 Java IO Stream Classes

### 21.3.1 Classification of ByteStream Classes

### 21.3.2 Classification of CharacterStream Classes or Text Classes

### 21.3.3 DataInput and DataOutputStream FileInput and FileOutputStream

### 21.3.4 FileReader and FileWriter

## 21.4 IO Errors and Exceptions

### 21.5 FilterInputStream and FilterOutputStreams

### 21.6 Using BufferedInput and BufferedOutput Streams

### 21.7 Writing Primitive Data Types to File: DataInputStream/DataOutputStream

## 21.8 File Class

## 21.9 Random Access Files

## 21.10 Serialization of Objects and Object Streams

### 21.10.1 Serialization

### 21.10.2 De-serialization

## 21.11 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

## 22 Networking in Java

### 22.1 Introduction

### 22.2 Basics of Networking

#### 22.2.1 TCP/IP

#### 22.2.2 User Data Gram Protocol (UDP)

### 22.3 Internet Address

### 22.4 URL and URL Connection

#### 22.4.1 URL Connection

### 22.5 TCP/IP Sockets

#### 22.5.1 ServerSocket Class

#### 22.5.2 Server and Socket for Communications: How to Set them to Work?

#### 22.5.3 Client and Socket for Communications: How to Set them to Work?



## 22.6 Client Server Program

### 22.6.1 Client Server Two-way Communication Program

### 22.6.2 Multiple Clients and Server Programs Using Multithreads

### 22.6.3 Client Server Program for File Download from Server

## 22.7 Summary

### *Exercise Questions*

#### *Objective Questions*

#### *Short-answer Questions*

#### *Long-answer Questions*

#### *Assignment Questions*

#### *Solutions to Objective Questions*

## 23 Graphics Using Swing Components and Applets

### 23.1 Graphics Programming and Applets Introduction

### 23.2 Hierarchy of Graphics Software

### 23.3 Containers in Swing

#### 23.3.1 JFrame Class

#### 23.3.2 JPanel Class

### 23.4 Display Widgets in Swing

23.4.1 JLabel Class

23.4.2 JButton

23.4.3 JCheckBox

23.4.4 JComboBox

23.4.5 JMenu Class

23.5 Layout Managers

23.5.1 Border Layout

23.5.2 Grid Layout

23.5.3 Flow Layout

23.6 Event Types and Event Listeners

23.6.1 Action Event Listener

23.6.2 Item Event Listener

23.6.3 Keyboard Event Listener

23.6.4 Mouse Event Listener

23.7 Applets

23.7.1 Concepts of Applets

23.7.2 Life Cycle of an Applet

23.7.3 Creating an Applet - HelloWorld Applet

23.7.4 JApplets - Applets with Swing Components:  
Better Look and Feel

23.7.5 Applets vs. Stand-alone Applications

23.7.6 Passing Arguments to Applets

23.8 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

24 Collections and Software Development Using Java

24.1 Introduction

24.2 Collection Framework of Java

24.3 Collection Interface

24.4 Set Interface

24.5 Iterators

24.6 List Interface

24.7 Collection Algorithms

24.8 Map Interface

24.9 Collection View of Map

24.10 JDBC Database Connectivity

24.11 Access Database Records Using MS Access Database

24.12 Access Database Records Using MySql Open Source Database

24.13 Access Database Records Using Oracle Database

24.14 Prepared Statement and Callable Statements

24.14.1 PreparedStatement

24.14.2 Callable Statements

24.15 Servlets

24.15.1 Servlets API and javax.servlet Package

24.15.2 HttpServletRequest

24.15.3 HttpServletResponse

24.15.4 HttpServlet Class—Get Request with Data from Client

24.15.5 HTTP Post Requests

24.16 Java Beans

24.17 Summary

*Exercise Questions*

*Objective Questions*

*Short-answer Questions*

*Long-answer Questions*

*Assignment Questions*

*Solutions to Objective Questions*

*Appendix A*

*Appendix B*

*Appendix C*

## About the Author

**Ramesh Vasappanavara** obtained his bachelor's degree from the University College of Engineering, Andhra University (1972–1977) and his master's degree from the Indian Institute of Technology Kharagpur (1982–1984). He obtained Ph.D. in Computer Science and Engineering from Jawaharlal Nehru Technological University Hyderabad. He has held several senior positions in Naval R&D and was a professor and the director of reputed engineering institutions such as Gayatri Vidya Parishad College of Engineering; Galgotia College of Engineering, Delhi; Indo-German Institute of Advanced Technology, etc. He is currently working as Director and Professor at SGIT Delhi, a prestigious engineering institution.

His professional interests include Grid Computing, Algorithms, and Embedded

Systems. His passion is to develop young minds in to original thinkers who can then provide innovative solutions to real-world problems. He can be reached on [ramesh.vasappanavara@gmail.com](mailto:ramesh.vasappanavara@gmail.com).

**Anand Vasappanavara** obtained his bachelor's degree from Indian Institute of Technology Madras (1999–2003) and his master's degree from the Indian Institute of Science, Bangalore (2005–2007). He has worked for Tata Motors, Pune, as a design Engineer between 2003 and 2005, where he developed a controller for a hybrid car for Telco. Since 2007, he has been working at Shell, as a Process Control Technologist.

His professional interests broadly include Energy, Automobiles and Control Systems. He envisions a world where energy and knowledge are available in abundance, which can result in real empowerment of the people. He can be reached on [vasappanavara@gmail.com](mailto:vasappanavara@gmail.com)

**Gautam Vasappanavara** obtained his bachelor's degree from the University

College of Engineering, Andhra University (2000–2004) and his master's degree in embedded systems from Birla Institute of Technology and Science, Pilani (2005–2007) as Phillips Research Scholar. He has worked for GE Controls, Hyderabad, and Samsung Electronics, Bangalore, as Lead Engineer in Wireless Technologies. Presently, he is on a sabbatical as he is pursuing a Management Degree from the Indian School of Business, Hyderabad. His passions include energy conservation, using technology as a platform for societal transformations and knowledge sharing. He can be reached on [gautam.vasappanavara@gmail.com](mailto:gautam.vasappanavara@gmail.com)

Web site: [www.vasappanavara.org](http://www.vasappanavara.org)



*Dedicated to*  
**Sri. M. Venkat Ratnam and Smt. M.**  
**Saraswati**  
*For holding our hands and making us what*  
*we are today!*  
&  
**Smt. V. Usha Ramesh**  
*For your extraordinary contributions to*  
*our family.*

# Preface

Welcome to the world of object-oriented programming using C++ and Java. Get ready for an exciting tour!

While teaching OOPs, C++ and Java to graduate and postgraduate students and practicing professionals from the industry for several years, our experience shows that:

- There are good programmers both in C++ and Java but very few among them understand the underlying OOPs concepts.
- The text books available treat C++, Java and OOPs concepts as different and separate and students get exposed to these concepts in a compartmentalized manner.
- The study of OOPs languages, analysis and design and software development are all treated separately thereby sacrificing the integrated OOPs approach.
- C++ is a highly evolved language and is used in systems development with advanced features like templates and containers. Most of the text books handle C++ as a language and not as a tool to implement OOPs concepts.
- Java is a vast and widely used network language with diverse OOPs concepts, networking programming and

for software development using tools such as JDBC, Servlets and Graphics Programming and Java Beans. A majority of text books cover the core Java adequately but they grossly fall short in providing implementation details.

- There is no single text book that covers the syllabi of all major Indian universities concerning OOPs, C++ and Java.

This book precisely comes to your aid in these critical aspects.

Although knowledge of C language is definitely a plus point, we assume no background knowledge. You can straight away master C++ and/or Java. The approach adopted in this text is to make students learn by practice and examples.

Each concept is explained with code snippets in both C++ and Java. The distinctive feature of these chapters is that there are numerous running examples, more than 600 of them, and case studies like College Administration, Internet Banking, e-shopping, Library Acquisition, etc. which students can easily correlate and learn using the underlying concept and theory. Unified Modelling Language (UML) notation is used throughout the book. At the end of each

chapter, we have provided objective questions, short- and long-answer type of questions, several exercise problems with solutions and assignment questions.

All the programs have been tested on Linux, VC++ and TC++ platforms. The development platforms we have used to run Java programs are eclipse, NetBeans and JDK1.6. The databases used are of industry standard, such as Oracle, MySql (Open source) and widely available Microsoft Access databases. The code and instructions are elaborate and are designed to provide solutions quickly and enhance learning experiences.

### Online Web Resources

Online Web resources are available on the book's companion Web site [www.pearsoned.co.in/rameshvasappanavara](http://www.pearsoned.co.in/rameshvasappanavara). They include PowerPoint slides, video lectures, an additional chapter, worked-out examples and program files that include C++ and Java

codes for all examples and exercise problems.

**PowerPoint slides:** All chapters have a PowerPoint slide highlighting key concepts discussed in that chapter.

**Video lectures:** Audio-visual lecture slides help students understand the concepts better.

**An additional chapter:** A chapter entitled “Standard Template Libraries (STL) and Containers” is available on the Web site.

**Worked-out examples:** Additional problems have been solved.

**Program files:** C++ and Java codes for all examples and exercise problems have been provided.

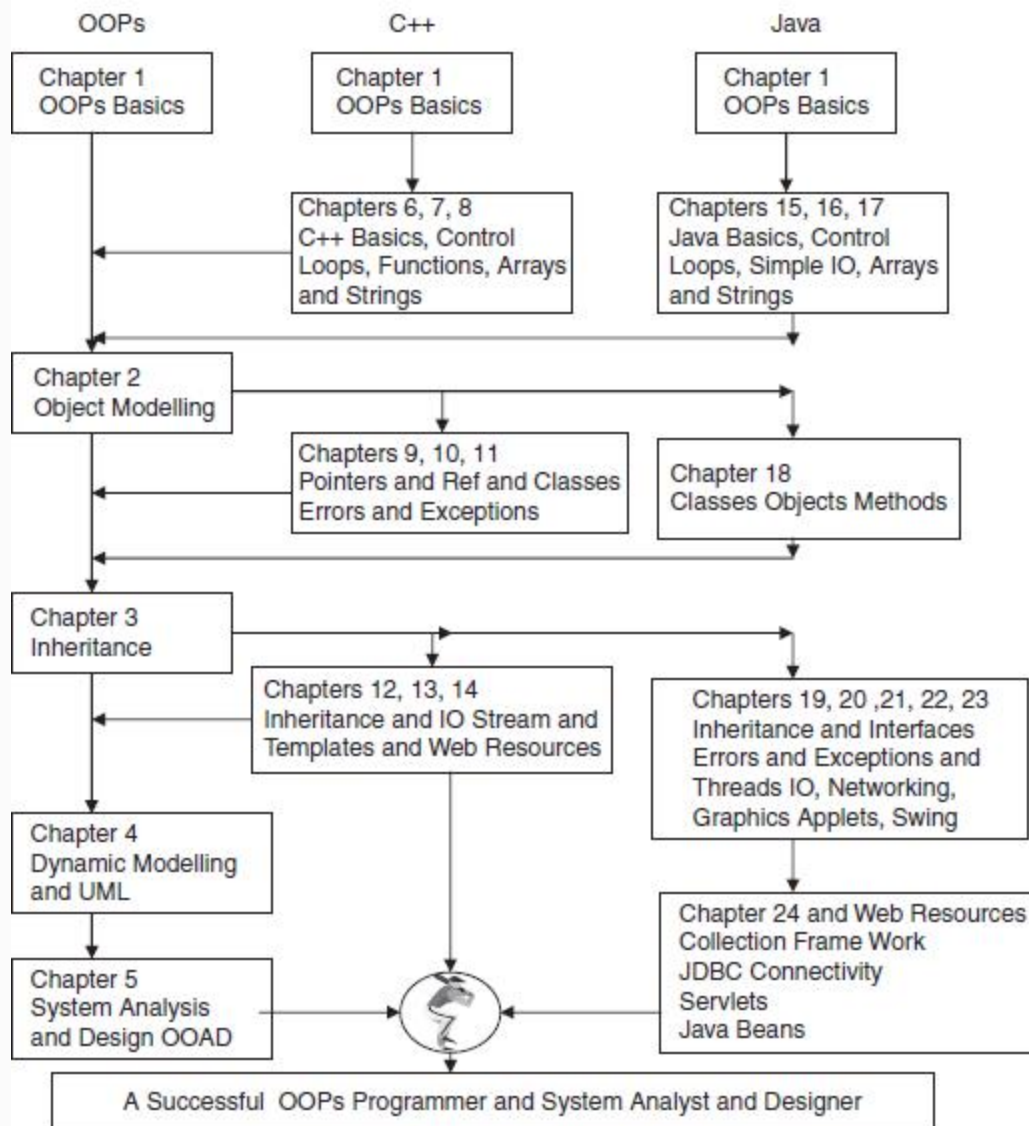
## Feedback

Utmost care has been taken in writing this book to make it free of errors. However, should you come across any error, please do

not hesitate to contact us. Your suggestions and feedback are welcome.

# How to Use This Book and Web Resources

The content and the best way to use this book are presented in the flow chart below:





# 1

## Object-Oriented Programming Basics

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to understand*

- Programming concepts.
- Software development paradigms.
- OOP paradigms and extendibility features offered by OOP languages.

### 1.1 Introduction

In order to solve a problem, you need to know the “method/procedure” or you

need to have the “know-how”. In computer parlance, we call this an algorithm. An algorithm or a program is a sequence of steps to be followed, which leads to a desired output. The sequence of steps can be called a procedure, and, in turn, a group of procedures can be termed as a program. For solving a simple problem, a sequence of steps is sufficient, but if the problem is complex, a programming environment that includes well-integrated and cohesive programming elements, constructs and data structures is required. A paradigm can be thought of as a style of programming, which involves elements such as functions, data and data structures. Several programming paradigms have been used successfully. The structured programming paradigm with C as implementing language, in which the programmer breaks down the task to be accomplished into subtasks and specifies a step-by-step procedure or algorithm to achieve this task, has been very successful and popular amongst programmers, owing largely to its reusable components in the form of procedural calls and its ability to

manipulate memory and thus handle hardware integration.

Low productivity of programmers, due to factors such as changing user needs, complexity of projects and non-availability of extensibility and reuse features, causes projects never to be completed on time. Even if they are completed and implemented, they are of little use. A structured programming environment with limited reuse facility afforded by procedure call and lack of extensibility features, i.e. inability to derive new data types from existing ones, is not optimally suited to handle challenges thrown by complex projects such as fast-paced hardware and software changes. Object-oriented programming (OOP) promises extensive reusability features through inheritance, library functions in the form of standard template library (STL), etc. C++ is one of the most powerful OOP languages that supports both structured as well as OOP paradigms.

## 1.2 Programming Concepts

Prior to the invention of C and C++, programming languages could be broadly categorized as follows:

**High-level languages** such as FORTRAN, Cobol and Pascal, catered to programmers with friendly features such as English-like coding languages. In these languages, the code written by programmers is compiled and converted into an object code. The implementation and hardware details such as addresses are hidden from the users. Hence users can code the application software for scientific and business communities. Cobol is a language widely used in the business world. FORTRAN and Pascal, on the other hand, are languages that are used by the scientific communities.

**Low-level languages** such as assemblers require extensive knowledge of hardware, addressing mechanisms and operating systems. Accordingly, these types of languages are used by experts, academicians and system developers. Most of the interfaces involving hardware integration

with software are written using these languages.

**Middle-level languages** such as C. These languages combine the strengths of both high-level and low-level languages.

- Code is almost English like. It is easy to understand and write programs in this language.
- It has the ability to handle hardware as many of its commands support the direct handling of underlying hardware such as memory and hardware devices.
- Both business programming and scientific calculations can be easily handled.

## 1.3 Programming Paradigms

Paradigm is a style of programming language. It is only a style and NOT a language. It is the manner in which programming elements such as functions, objects and variables are exploited to produce the desired output.

### *1.3.1 Structured Programming Paradigm*

Procedure-oriented programming such as C uses procedure calls. The main task is divided into subtasks with inputs and functions to achieve the desired result. The

data is forwarded to procedures through arguments. In this paradigm, the procedure is important and the data is shared by procedures.

Algorithm or program or procedure is a sequence of steps to be followed, which when followed leads to a desired output. A group of procedures can be termed as a program. A procedure is also known as a routine, subroutine, function or method. A program or a procedure under execution can call any other procedure.

Accordingly, using the procedure-oriented language, the programmer breaks down the task to be accomplished into subtasks and specifies a step-by-step procedure or algorithm to achieve this task. A procedure-oriented language like C provides the statements to code the procedure specified by the task. Procedure-oriented languages have several advantages, as follows:

- **Modular:** Each subtask can be developed as a module. A module takes arguments as inputs and delivers outputs as return values.
- A module which is independent and uses standalone codes avoids coding pitfalls such as GOTO and Jump and thereby makes it easy to maintain the code.

- Reusability is ensured. The module can be called by any other module during execution. All reusable codes can be maintained as libraries.
- Strong modularity and reusability features afforded by C make it the most suitable language platform to develop complex programs and projects.

A procedure-oriented language is also known as a "structured language" because it allows procedure calls, which in turn allows a programmer to break the main task into subtasks and execute each subtask through procedure calls.

### *1.3.2 C: A Workhorse that Works Well*

The C language was developed by Dennis Ritchie of Bell Laboratories in 1972 for the UNIX operating system. It had originally been developed as a system programming language, but it soon caught the imagination of application developers and became the industry standard.

C code is compiled into a machine code using a straightforward compiler and hence does not need much run-time support. Hence, C found acceptance and popularity

amongst assembler programmers for their system programming requirements.

C is an almost machine-independent language. A compiled code on a machine can run on several different platforms. The salient features of C are as follows:

- All executable codes are included in functions only. Inputs are through arguments and outputs are through return values.
- All parameters are passed either as pass by value or by reference using pointers.
- Structures allow heterogeneous data to be grouped into one single data unit.
- Memory access through pointers.
- Library routines for IO, string manipulation, and mathematical calculations.

Thus, a structured programming language like C has been able to satisfy all programmers, with its ease of programming, reusable features through procedure calls and ability to deal with hardware integration.

Then why do we need to learn objective-oriented programming?

### *1.3.3 Where is the Problem?*



Problems arise due to fast-paced changes. Changes occur at such a rapid rate that projects and programs developed based on inputs at a particular instance or time are no longer valid. Thus, the project developed is confined to the shelves of a departmental library, never to be implemented.

This is especially true for software projects, wherein the latest software programming paradigms and hardware are required.

User requirements change during the development of project. In a complex project, changes in user requirements imply rework.

The US Department of Defense (DoD) has found out that a majority of the projects are out of date and cannot be implemented.

#### **1.3.3.1 Complexity of Problems**

Initially, computers were being exploited for basic business computing such as for payrolls, student or employee records, and scientific calculations. These are all simple algorithms involving input, process and

output and could easily be handled by structured programming.

The complexity of problems, however, has grown rapidly and modern-day software projects involve multilayer inputs and communication with local and distant processors, networks, etc.

#### **1.3.3.2 User's Needs**

Analysis, design and development are scheduled after a thorough study of user requirements during the initial stages of project implementation. It is correct to state that requirements are specifications that decide every stage of software design and development. Hence they are “frozen” before commencing the development phase.

But on the ground, it is important for operational managers, e.g. airport operations managers, to implement changes to ensure the smooth functioning of day-to-day operations at airports. Thus, it is only natural for them to expect these latest changes to be incorporated in the software being developed.

There go your initial estimates!  
Developers have to rework the design,  
depending on the nature of changes sought.

It is estimated that rework due to changes constitute about 40% of time and effort estimations and have a significant effect on project overruns.

#### **1.3.3.3 Low Productivity**

Extensibility and reuse are two features that make a programmer productive. In C, we have struct and unions as user-defined data types. The facility to define new data types based on already-defined ones is not available.

Changes brought in result in changes in several procedures and often result in recompiling the source.

Extension of the data type through inheritance is not possible. Programmers have to be content with procedure calls.

#### ***1.3.4 Object-oriented Programming Paradigm***

Object-oriented programming such as C++ uses objects and interaction amongst them

through invoking member functions. The features include data hiding, data abstraction, encapsulation, inheritance and polymorphism.

With the advent of UNIX and communication hardware and software from 1969 onwards, the complexity of both hardware and software increased and software project development could not keep pace. As a result, the quality of programs suffered.

The developers at Bell Laboratories and the DoD started to look at alternative paradigms wherein reusability and extensibility was a strong feature. Further, the specifications laid insisted on data primacy rather than process primacy.

The object in OOP paradigms contains member data and all functionality within itself to achieve the desired result. The class and the object of the class carry their own operators to achieve a particular functionality. For example, an object may carry an overloaded operator, `>>`, to achieve input operations. Objects communicate with other objects by passing messages.

# Salient Features of the OOP Paradigm

- Data primacy and *not* procedure primacy.
- A task is divided into objects.
- Each object is an independent machine with its own data structure and member function in its own memory location.
- Methods and data are tied together in an object.
- External functions cannot access data.
- Objects can send messages to another cooperating object.

## 1.4 History and Development of Object-oriented Languages

Smalltalk was the first OOP language to be developed. Currently C++, Java and C# are the languages that use OOP paradigms and are industry favourites.

### 1.4.1 C++

C++ was developed by Bjarne Stroustrup in 1979 at Bell Laboratories. C++ is a sequel to the C language, with additional features. These features include virtual functions, function name and operator loading, references, and free space memory along with several features on reusability and extendibility through inheritance, templates,

etc. Exception handling and a well-developed standard template library are two of the most advanced features available in C++.

### *1.4.2 Java*

Java was developed by James Gosling at Sun Microsystems in 1995. It is similar to C & C++. The main advantage of Java is that it is hardware independent. Any code is compiled into **bytecodes**. The bytecode compiled is stored in a class file that can run on any Java virtual machine (JVM).

JVM takes the bytecode and converts it into an object code, which in turn is used by a local machine to run and produce the desired result.

Java 1.0 was released in 1995. Sun Microsystems called it "Write Once, Run Anywhere". Indeed, with most of the browsers such as IE explorer and Netscape Navigator supporting Java 1.0, Java soon became the industry standard. Java applets that can run on web browsers became popular too. In 1998, Sun Microsystems

released Java 2 with specialized configurations to suit different types of platforms, such as J2EE for enterprise applications, J2ME for mobile computations and J2SE for standard editions. These have been renamed JavaEE, JavaME and JavaSE, respectively.

### *1.4.3 C#*

C# was developed by Anders Hejlsberg for Microsoft. It combines the best features of existing OOP programming languages of C++ and Java. It has been developed as a powerful and versatile programming language that is fully object oriented.

## *1.5 Software Development Methodologies*

Software development is an engineering activity involving initiation, analysis, design, development, and implementation and delivery phases. The principles and guidelines for developing projects using software engineering (SE) practices are enshrined by the SE branch of engineering.

What is a methodology? A methodology can be considered as a style of solving an SE (project) problem from the best practices prescribed by the SE domain. For example, we can adopt structured analysis and design (popularly known as SAD in earlier times), which relies on procedure-oriented languages such as C, Pascal, etc.

Alternatively, we can also adopt object-oriented analysis and design (OOAD), wherein objects comprising member data and member function achieve the desired output through interaction amongst them.

## 1.6 Need for Objects

In OOP paradigms, data structures that manipulate data and functions that use data are not separate. OOP combines both of these into a single ***thing, also known as object***.

Objects bind the data and the functions that operate on the data together by a concept called encapsulation, thereby avoiding unauthorized data access and maintaining tighter control over data.



Object is a run-time instance and communicates with other objects by passing arguments and thereby achieves the result. Indeed, objects and communication between objects is the methodology adopted by object-oriented programs.

## 1.7 Object-oriented Language Features

OOP style can be of two types, i.e. object-based programming and object-oriented programming. In OOP paradigms, object is primacy.

### *1.7.1 Object-based Programming*

To qualify as an object-based language, a language must support the following features:

- Data encapsulation
- Data hiding
- Operator overloading
- Initialization and automatic clearing of objects after use

### *1.7.2 Object-oriented Programming*

In addition to the object-based features mentioned above, OOPs must support

extensive reusability and extendibility features such as:

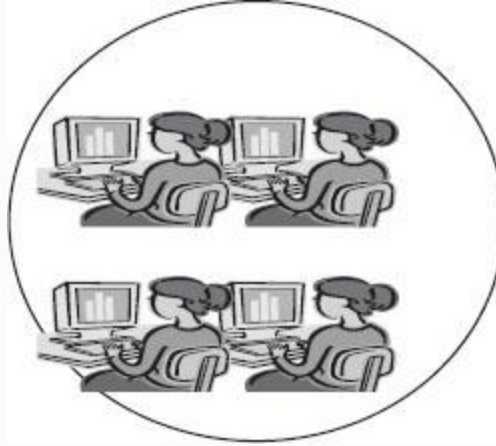
- Inheritance
- Dynamic binding
- Extensive and well-defined standard template library

In the succeeding sections, we study the basics of OOP language features. C++ and Java are the most powerful OOP languages that support both structured as well as OOP paradigms.

## 1.8 Definition of OOP Language Classes and Objects

Look around. You will notice several objects (things) such as pens, tables, laptops, students, etc. Figure 1.1 shows objects of students in a class. Each **thing** has attributes, also called characteristics, like colour, height, weight, age, etc. The attributes belong to the objects. For example, an object called student can have attributes like name, number, marks, grades, etc. Attributes are also called the **state** of an object. Most objects have their own behaviour. For example, an object

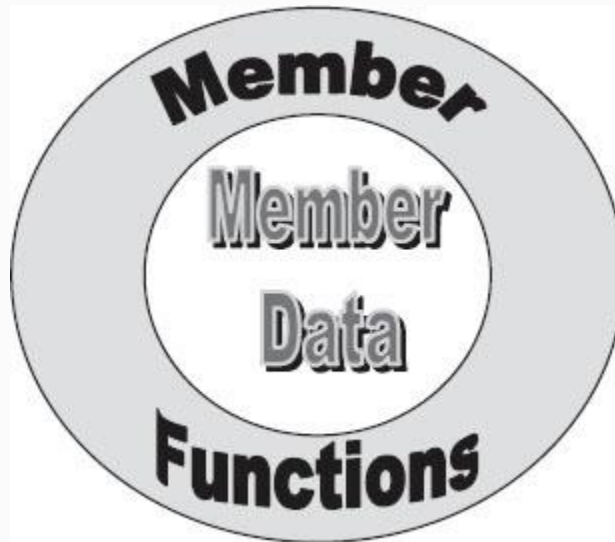
called student can have behaviours like, play, sing, learn, etc.



**Figure 1.1** Student objects

So when we refer to an object called student, we refer to both **attributes (state) and behaviour**. Figure 1.2 shows attributes and behaviour. We model an object by a rectangular box, as shown in Figure 1.3. It contains the name at the top, followed by a list of member functions and member data. It is customary to show the member data as **private** and functions that manipulate these data as **public**. This is a security feature, enshrined in OOP

languages such as C++ as **data hiding and encapsulation**, to prevent unauthorized access to objects' data by external functions.



**Figure 1.2** Object: member data and functions

Student
<pre>// member functions public :     void Play()     void Learn()     void GetData()     void printData()</pre>
<pre>// member data private :     char name     int rollNo;     float total;     char grade;</pre>

**Figure 1.3** Object model

How do we represent the object?

### *1.8.1 Attributes and Behaviours of Objects*

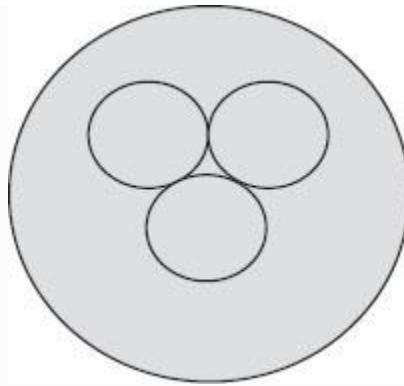
There may be several instances of an object. For example, an object called student as shown above can have 60 instances, implying that there are 60 students in a class. Therefore, each instance of an object can have its own data, such as its own number, name, marks, attendance, etc. We can thus say that the attributes, i.e. data, of an object are owned by that instance of the

object. Hence, **attributes are also called *states of an object***.

All student objects will have the same functionality such as `Play()`, `Learn()`, `GetData()`, `PrintData()`, etc. Thus, when we say object, we mean attributes and behaviours.

### *1.8.2 Class*

**Class:** A class is a collection of objects and can also be defined as an array of instances. Figure 1.4 shows student class with three instances. It can also have member functions and member data. Here unlike arrays, a class can have different data types as its elements.



**Figure 1.4** Student class with three instances of student object

Class defines abstract, i.e. hidden, characteristics of an object, including attributes and behaviours. A class called student can be viewed as a factory producing instances of object student that have different attributes (i.e. individual data) and a common functionality (i.e. common functions).

Attributes and functions provided by a class are called member data and member functions. In Java, member functions are called methods.

### *1.8.3 Encapsulation*

By now we understand that object means attributes and functionality. Class can contain several instances of an object.

We also understand that in object-oriented programming, it is a data primacy language, i.e. data is important and functions are not important. As per memory mapping used by C++ and other OOP languages, data is stored in data areas such as stack and free space, functions are stored in code areas and there is a need to maintain strict control over the accessing of data by functions.

OOP languages achieve this control by using the encapsulation feature.

*Encapsulation is a binding member data and calling function together with security classification, so that no unauthorized access to data takes place.*

#### **1.8.3.1 Security or Access Privileges**

C++ and Java depend heavily on access specifiers to maintain data integrity. The security access specifiers are public, private, and protected.



- **Public:** Member functions and data, if any, declared as **public** can be accessed outside the class member functions.
- **Private:** Member data declared as **private** can only be accessed within the class member functions and data is hidden from outside.
- **Protected:** Member data and member functions declared as **protected** is private to outsiders and public to descendants of the class in **inheritance** relationship. You will learn more about this in the inheritance chapter under C++ and the Java chapter that follow.

Encapsulation can now be defined as:

*Binding together the member functions and member data with access specifiers like private, public, and protected into objects by the class.*

*A class therefore allows us to encapsulate member functions and member data into a single entity called an object.*

#### *1.8.4 Data Hiding / Data Abstraction*

It is customary to declare all member data as private only. But the data declared as private is hidden and cannot be accessed by anyone. This feature is called data hiding or data abstraction. But how can this be achieved? You can access this only through public

member functions. There is no other way. It is comparable to the case where even the chief librarian of a university cannot take home books unless he uses the access card supplied by the library.

### *1.8.5 Function Overloading*

When a single function can do more than one job, overheads of the compiler get reduced. For example, if one wants to compute the area of a circle, the surface area of a football or the surface area of a cylinder, how many functions do we have to write? Normally, three functions. But in OOP languages like C++, one function, overloaded to perform all the three jobs, is sufficient.

Overloaded functions decide which version of the code is to be loaded into primary memory, depending on the argument supplied by the user.

Why is overloading important? Functions have to be compiled and loaded into primary memory. If a function is NOT overloaded, all the functions have to be loaded and linked at

the time of compilation. Users may or may not use all the functions loaded, thus wasting the primary memory and making it unavailable to solve complex problems requiring more primary memory.

As an example consider the following overloaded function:

---

```
// single argument. Same function name
void FindArea( float r ){ return (
2*3.14158*r*r ) ; }
// two argument same name. find
surface area of the cylinder
void FindArea( float r , float h ){
return ( 4*3.14158*r*h ) ; }
```

---

### *1.8.6 Operator Overloading*

Predefined operators such as  $+$ ,  $-$ ,  $/$ ,  $*$ , etc. are defined and provided by the compiler to work on intrinsic data types such as `int`, `char`, `double`, etc. We have also seen while discussing the concept of function overloading that if a function can perform more than one task, we can call this overloading. This is an efficient way of utilizing the scarce resource like primary

memory. Operator overloading also improves the efficiency and throughput of the program by conserving the primary memory of C++.

If we can use these predefined operators to work on user-defined data types such as classes, we would call it operator overloading. For example, consider a predefined operator + and its normal operation

---

```
int x = 10, y = 20;  
int z = x + y ; // contents of x and y  
are added and placed in z
```

---

Now consider two complex numbers in polar form of representation:

---

```
Polar v1(25.0,53.50); // magnitude and  
angle theta  
Polar v2(5.0, 45.00);
```

---

If we can write

---

```
Polar v3 = v2 + v1;
```

---

It means we have overloaded the + operator.

## 1.9 Extendibility and Reusability of OOP Paradigms

The extendibility and reusability of OOP paradigms are responsible for making the OOP languages versatile and powerful. There are several tools provided by OOP languages to ensure these features:

- Containment
- Inheritance
- Virtual functions and abstract data types (ADTs)
- Standard template libraries (STL)

We will briefly describe the above features in the succeeding sections. Their implementations will be explained in the C++ and Java chapters in Parts 2 and 3.

## 1.10 Extending / Deriving New Classes

We can derive a new class based on the existing class. This is possible through the inheritance and containment property afforded by OOP languages. Containment is

class within a class, i.e. containment ensures all the member functions and member data of a contained class to the class containing the class. Inheritance, on the other hand, provides access to member data and member functions declared as protected to the descendant class.

### *1.10.1 Containment: Class Within a Class – Container Class*

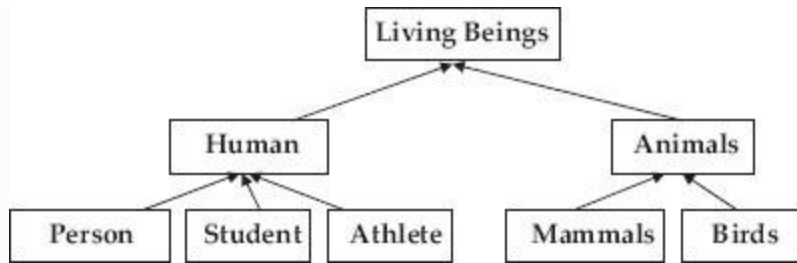
Container class is one of the techniques provided by C++ to achieve reusability of the code. Container class means a class containing another class or more than one class. For example, a computer **has** a microprocessor in it. Further, a student class can have a data member belonging to a string class. We would then say that a string class is contained in a student class. In other words, it can also be called **composition or aggregation** of a string class.

The advantage of containment is that we can use a string class within a student class to store the details of name, address, etc. belonging to a string class. Similarly, if we

define a class called date with day, month and year information, we can define the object of date within a class called student and define date-related data such as DOB, DOJ, etc. Therefore, **composition is a "has" type of relation.**

### *1.10.2 Inheritance and Class Hierarchy*

Reusability of a code is one of the strong promises made by C++ and inheritance is the tool selected by C++ to fulfill this promise. The concept of inheritance is not new to us. We inherit property, goodwill and name from our parents. Similarly, our descendants will derive these qualities from us. Refer to the inheritance class hierarchy shown in Figure 1.5.



**Figure 1.5** Inheritance hierarchy

Humans and animals derive qualities from living beings.

Professionals, students and athletes are derived from humans. We say these three categories have inherited from humans. Class human is called base class and students and athletes are called derived classes. What can be inherited? Both member functions and member data can be inherited.

Have you noticed the direction of the arrow to indicate the inheritance relation? It is pointed upwards as per modelling language specifications.

Inheritance specifies an **is** type of relation. Observe that a student is a human. Similarly, a mammal **is** an animal. Human is



a **base class** and student is a **derived class**. Derivation from base class is the technique to implement the **is** type of relation. A base class can have more than one derived class.

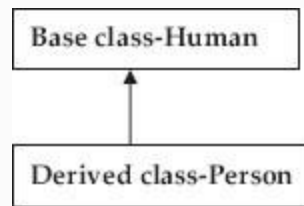
When do we use inheritance? You inherit so that you can derive all the functionality and member data from base class, and derived class can add its own specialized or individualistic functionality. For example, athlete derives all functionality and attributes of human, and, in addition, adds sports and athletic functionality and attributes on its own.

Inheritance is a powerful tool in the hands of a programmer to define a new class from existing classes.

### *1.10.3 Single and Multiple Inheritances*

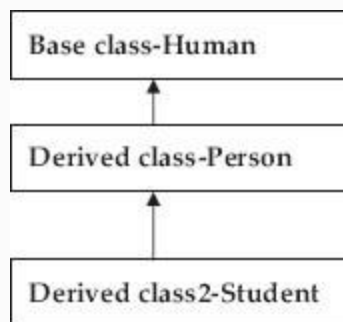
Inheritance means the ability to derive a new descendant class from a base class, with additional functionalities.

In Figure 1.6a, we have shown **single inheritance**, wherein the derived class person inherits from the base class human.



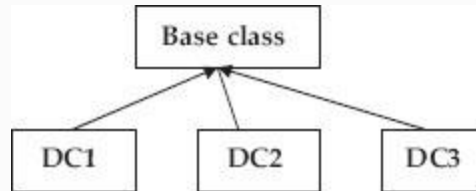
**Figure 1.6a** Single inheritance

In Figure 1.6b, **multilevel inheritance** is depicted, wherein student inherits properties from person, which in turn inherits properties from the base class human.



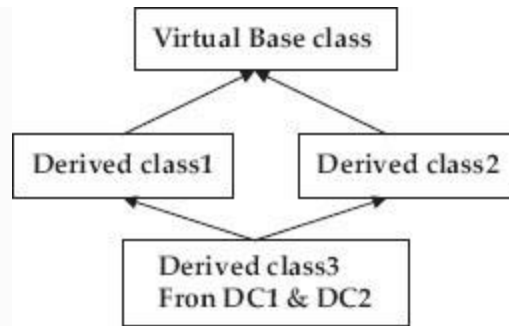
**Figure 1.6b** Multilevel inheritance

In Figure 1.6c, we have shown the **hierarchical inheritance** involving one base class and three derived classes.



**Figure 1.6c** Hierarchical inheritance

Figure 1.6d depicts **multiple inheritance**. In this, the base class is called virtual because a single base class is being used by two derived classes, namely derived class 1 (DC1) and derived class 2 (DC2). Further, you can see that derived class 3 (DC3) inherits from both DC2 and DC3. This type of inheritance is also called hybrid inheritance as there is a single derived class DC3 from two base classes DC1 and DC2.



**Figure 1.6d** Multiple/hybrid inheritance

## 1.11 Virtual Functions

In the inheritance relation, the derived class gets access to protected member data through public accessory functions, thereby achieving an important object-oriented programming facility called code reusability.

But what about object to base class using the member functions belonging to a derived class? This feature facilitates object to base class to execute any one of the derived class functions, depending on the users' choice at run time.

This means that object to base class is provided with a bridge to the selected derived class function and hence it can

execute the function. The bridge is made available when we declare a virtual function in the base class and when the derived class overrides the base class function.

If a function is declared as a virtual function in base class, we can execute an overriding function with the same name in the derived class with a pointer to base class. A pointer to base class is provided by a virtual function.

In summary, we can say that from base class we can execute any function with the same name as that of a virtual function in the base class, depending on the users' choice at run time.

## 1.12 Run-time Polymorphism and Dynamic Data Binding

Inheritance solves the problem of reusability, i.e. a derived class can access all the data members and function members of a base class that are declared as protected.

We have also learnt that a derived class can override the functions defined in a base class.

Further, we have seen while dealing with virtual functions that with a pointer to base class we can call the derived class object.

Combining the above two features of OOP languages like C++ gives us a powerful tool, called run-time polymorphism and dynamic binding.

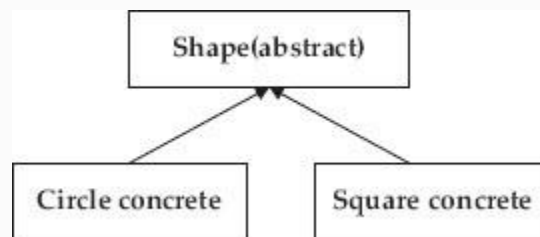
What this feature means to a programmer is that with a pointer to base class, we can decide at run time to call particular derived classes overriding a function. In other words, we can bind the overriding function from several of the derived classes with a pointer to base class. In the previous section, we have seen the working of virtual functions with which we can achieve binding of derived class functions with a pointer to base class.

### 1.13 Class as Abstract Data Type (ADT)

A class in object-oriented languages can also be called **abstract data type (ADT)**. The features of OOP languages such as run-time polymorphism and dynamic binding allow us to define a base class with virtual

functions with no implementation (called pure virtual functions) or with dummy functionality just to indicate to the user that implementation is by derived class. These classes are called ADTs because they hide the implementation.

Refer again to Figure 1.7. We will declare a base class (ADT) called shape. You will appreciate shape has no definite shape, no definite area or perimeter, and no specific draw routine. Shape will hence declare virtual functions with only names and no implementation details.



**Figure 1.7** Abstract Data Type (ADT)

We will derive classes like circle and square and provide the **solid** implementation of these virtual functions.

Solid means that all virtual functions will be implemented in each of the derived classes.

## 1.14 Standard Template Library (STL)

One way to describe object-oriented programming is a programmer's ability to use library facility through language. C++ has much developed standard library in the form of stream IO library that provides extensive functionality for input and output.

STL is the most significant addition to the library provided by C++. Using the module in STL, a programmer can write code using data structures and algorithm implementations provided in the STL.

Vectors, lists, queues, maps, etc. are some of the implementations of STL. We have presented a detailed description and procedures to exploit templates and STL in Chapter 16. Templates and STL are considered to be the most significant and important OOP features of C++. They are indeed tools of C++ to enhance the programmer's productivity. The basic idea is that if you use already developed code, albeit



by a manufacturer or a third-party supplier, you are using the **reuse** feature of OOP.

## 1.15 OOPS – Object-oriented Programming and Systems

From our long experience of teaching undergraduate and post-graduate students, it has been our experience that students are good programmers but lack adequate exposure to become good project team members or leads. Where does the problem lie?

Students need to be taught to think object-oriented programming and systems (OOPS) skills right from step 1. In this section, we explain the terms and terminologies connected to analysis and design using object-oriented technologies and the use of UML (Unified Modelling Language).

We have already discussed objects in detail. Objects are things you see everywhere around you, like pens, students, etc. There are living objects and inanimate objects. We have also understood that objects have

***attributes***, also called member data, and ***behaviour***, also called member functions.

In an object-oriented design (OOD) model, objects have a class relationship, i.e. objects having the same functionality characteristics are grouped into a ***class***. ***A class allows us to encapsulate member functions and member data into a single entity called an object.***

Objects communicate with other objects by invoking functions and passing arguments to the functions.

Classes have relationships with other classes. These relationships are called ***associations***.

There are also inheritance relationships, using which we can define new classes from existing classes. There are also multiple inheritance relationships.

We have also learnt that virtual functions coupled with inheritance gives us a powerful tool like a run-time polymorphism and allows us to define ADTs and interfaces. Thus, we can decouple interface and implementation.

Programming in C language revolves around functions. Invoke functions and get the desired results. In OOP methodology, the desired result is achieved by passing messages between objects, i.e. the objects communicate by invoking functions and passing arguments amongst themselves.

We can define all classes required to execute a large project into a single working space. This space can be called a package. Once included in the package, we can use all declarations of classes into other programs and projects, thus ensuring reusability of the code.

### 1.16 Object-oriented Analysis and Design (OOAD)

In order to prepare a solution, we must first understand the system. To understand the system we need to model the system. Only then can we ensure programs and projects run as desired. So whenever a problem is non-trivial and difficult to understand, we model the system to study the system behaviour.

In order to understand the system, user specifications are prepared by users and experts. The specifications are simple statements of problems in English. We need to analyse the specifications and follow object-oriented analysis and design principles. The best way to analyse and design non-trivial and complex systems is through modelling by using UML.

UML is a graphical and pictorial language used for analysis and design methodologies. It is called unified because it unifies all earlier object-oriented methodologies enunciated by three of the founding fathers of object-oriented technologies, Grady Booch, James Rumbaugh and Ivar Jacobson. UML was founded by a consortium of industries such as HP, IBM, Oracle, etc. under the aegis of a controlling group called Object Management Group (OMG). We will introduce all these concepts in the succeeding chapters under OOPS.

In the next chapter, we will deal with concepts involved in object modelling and UML.

## 1.17 Summary

1. Structured programming involves invoking functions and thereby solving the problem.
2. A class allows us to encapsulate member functions and member data into a single entity called an object.
3. Classes have relationships with other classes. These relationships are called **associations**.
4. The main advantage of Java is that it is hardware independent. Any code is compiled into **bytecodes**. **The bytecode compiled is stored in a class file that can run on any JVM.**
5. JVM takes the bytecode and converts it into an object code, which in turn is used by a local machine to run and produce the desired result.
6. OOP styles can be of two types, i.e. object-based programming and object-oriented programming. In OOP paradigms, object is primacy.
7. To qualify as an object-based language, a language must support the following features: data encapsulation, data hiding, operator overloading, and initialization and automatic clearing of objects after use.
8. OOP must support the extensive reusability and extendibility features such as all object-based properties, inheritance, dynamic binding, and extensive and well-defined STL.
9. Each **object** has attributes (states) and behaviours. **Attributes are also called the states of an object.**
10. **Class:** A collection of objects. A class defines the abstract, i.e. hidden, characteristics of an object including attributes and behaviours.
11. *Encapsulation is a binding member data and calling function together with security classification to ensure that no unauthorized access to data takes place.*

12. *A class allows us to encapsulate member functions and member data into a single entity called an object.*
13. Function overloading is the same function name but with different numbers or with different types of arguments.
14. A container class means a class containing another class or more than one class. In other words, it can also be called **composition or aggregation**.
15. In an inheritance type of relationship, we can derive a new class from existing classes.
16. Inheritance means the ability to derive a new descendant class from a base class, with additional functionalities.
17. If a function is declared as a virtual function in a base class, we can execute an overriding function with the same name in the derived class with a pointer to base class. A pointer to base class is provided by a virtual function.
18. A class in object-oriented languages can also be called an **ADT**.
19. The features of OOP languages such as run-time polymorphism and dynamic binding will allow us to define a base class with virtual functions with no implementation (called pure virtual functions) or with dummy functionality just to indicate to the user that implementation is by a derived class. These classes are called ADTs because they hide the implementation.
20. Encapsulation means binding data and code. When run-time polymorphism is being used, it is best to polymorphically decouple the encapsulation to prevent discrete code modules from interacting with each other.
21. A base class overridden function is automatically hidden and can be called explicitly by referring to the base class function.

# Exercise Questions

## *Objective Questions*

### 1. C++ supports

1. Structured programming
2. Object-oriented programming
3. Both A and B
4. None of these

### 2. C++ has been developed by

1. Gosling
2. Bjarne Stroustrup
3. Rambaugh
4. Grady Booch

### 3. Java has been developed by

1. Gosling
2. Bjarne Stroustrup
3. Rambaugh
4. Grady Booch

### 4. Java is

1. Hardware independent
2. Software independent
3. Language independent
4. Firmware independent

### 5. Java virtual machine (JVM)

1. Converts source code to object code
2. Converts source code to bytecode
3. Converts bytecode to object code
4. None of these

### 6. C# has been developed by

1. Anders Hejlsberg
2. Bjarne Stroustrup
3. Rambaugh
4. Grady Booch

7. In OOP paradigms, a task is

1. Divided into functions
2. Divided into objects
3. Divided into subtasks
4. Divided into classes

8. OOP paradigm is

1. Data primacy
2. Procedure primacy
3. Task primacy
4. Object primacy

9. The following is not part of an object-based programming paradigm

1. Data encapsulation
2. Data hiding
3. Operator overloading
4. Inheritance

10. Member data declared as private

1. Can be accessed by an outside class
2. Can be accessed only by inside class member functions
3. Can be accessed by descendant class members
4. None of these

11. Member data declared as protected

1. Can be accessed by an outside class
2. Can be accessed only by inside class member functions
3. Can be accessed by descendant class member functions
4. None of these

1. i
2. i and ii
3. ii and iii
4. i, ii and iii

12. Containment is a(n)

1. Has type of relation
2. Is type of relation
3. Has-is type of relation
4. None of these



13. Containment is also called

1. Inheritance
2. Aggregation
3. Composition
4. Class within a class

1. i
2. i and ii
3. i and ii
4. ii, iii and iv

14. Inheritance is a(n)

1. Has type of relation
2. Is type of relation
3. Has-is type of relation
4. None of these

15. Object-oriented programming achieves results by

1. Invoking functions
2. Objects invoking functions and passing messages between objects
3. Functions invoking objects
4. None of these

16. Classes having relationship with other classes are called

1. Inheritance
2. Association
3. Link
4. Mapping

17. UML stands for

1. United Modelling Language
2. Unified Modelling Language
3. US Modelling Language
4. None of these

18. UML unifies the approach

1. Structured analysis and design and OOAD
2. Models of Jacobson and Rambaugh
3. Models of Rambaugh, Jacobson and Grady Booch
4. a and c

### *Short-answer Questions*

19. What is structured programming?
20. What is object oriented programming?
21. What is object-based programming?
22. Distinguish between object and class.
23. What is encapsulation?
24. Explain data hiding.
25. Why are classes called abstract data type?
26. What is message passing in OOP paradigms?
27. Explain containment.
28. Distinguish between function overloading and operator overloading.
29. Explain the features of run-time polymorphism and dynamic binding.
30. What are virtual functions?
31. What is UML?
32. What does the term unified in UML mean to you?

### *Long-answer Questions*

33. What are the salient features of OOP paradigms?
34. Distinguish between object-based programming and object-oriented programming. (OOP style can be of two types, i.e. object-based programming and object-oriented programming. In OOP paradigms, object is primacy.)
35. Explain inheritance and class hierarchy.
36. Distinguish between different types of inheritance.
37. Why are virtual functions useful?
38. Explain STL in C++.

## *Solutions to Objective Questions*

1. c

2. b
3. a
4. a
5. c
6. a
7. b
8. a
9. d
10. b
11. c
12. a
13. d
14. b
15. b
16. b
17. b
18. c

# 2

## Object Modelling

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to*

- Understand object modelling.
- Use notations of UML to specify objects and classes, their notations, meta models and concepts like mandatory profiles, meta data, meta classes, etc.
- Learn to use class diagrams and object diagrams of UML.
- Understand the concept of links and hierarchy, polymorphism and abstract classes in OOPS language.

### 2.1 Introduction

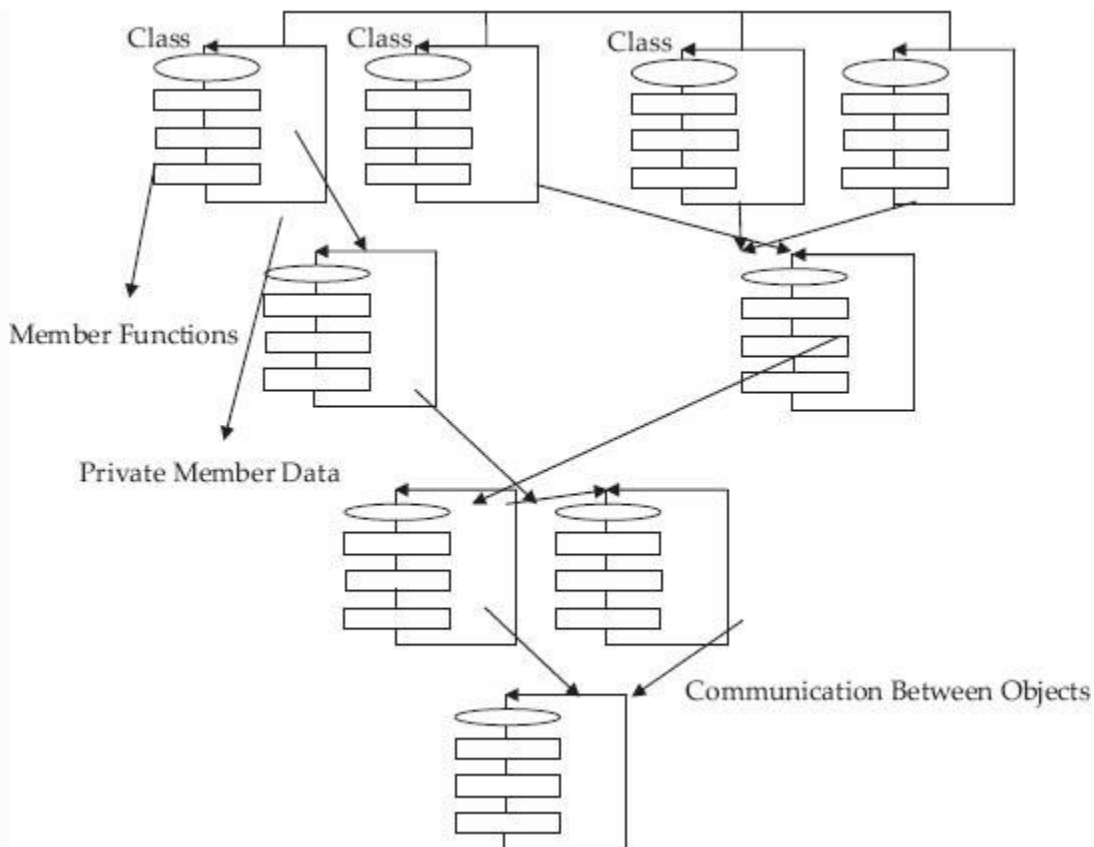
Object modelling encompasses all the elements that we studied in Chapter 1 regarding objectoriented (OO) methodology. The principles of object modelling are abstraction, encapsulation, modularity, hierarchy, typing, concurrency and persistence.

In conventional programming styles, we conceive the problem and subdivide the main task into subtasks and develop functions or methods to solve these subtasks. We call this paradigm procedure-oriented programming style. In this paradigm procedure is primacy and data that is acted on is stored at a different location in computer memory called the code area. In object modelling, the object embodies data and the functions that act on the data as a single entity. To meet users' requirements, the specifications are prepared by experts as simple statements of problems in English. In order to analyse these specifications, we need to study the vocabulary of the problem statements in terms of objects (nouns) and classes, and

relationships between classes appearing in the problem domain.

## 2.2 Object Model

The basic building block in an OO language is a module. What is a module? Modules can be defined as logical collections of classes and objects. If functions are verbs, then objects are nouns. Objects encompass the functions and member data. The building block in an OO methodology is shown in Figure 2.1, where each module comprises a logical collection of classes which in turn contain objects. Objects call member functions from other objects, a phenomenon, we called it as message passing and thereby achieve the result.



**Figure 2.1** Object-oriented programming methodology –  
Module

We observe that our building blocks are not a function or a procedure or an algorithm, but rather the classes and objects and messages passing between them. Observe from the above diagram that individual objects are responsible for their own member data and a set of functionality.

Thus objects store data and also functions necessary to update the data and do the necessary computations on the stored data. In addition, cooperating objects communicate with each other by invoking functions and passing information amongst them.

Object modelling is not only about mere object and contained data. Many a times, the system requires that *links* between data belonging to different objects be defined to achieve some global functionality. The object model must contain the relationship between the classes and data belonging to objects belonging to different classes. This will be decided by inheritance relationship between the classes.

The links between objects are message calls. They are requests sent to receiving objects to execute the function belonging to it with the parameters sent by requesting objects.

We are now ready for the definition of OO programming ***as a collection of cooperating objects which are in turn instances of a class. The cooperating***



***classes have a hierarchy of relationships defined by inheritance relationship.***

Notice that OOP language is about cooperating objects and not procedures and functions. Further notice that objects are instances of classes. Classes themselves are linked to other classes via the inheritance relationship. Therefore, we can say that OOP language means: ***objects, classes, inheritance.***

## 2.3 Object-oriented Design

At the end of analysis and design, we ultimately develop the code. The code has a structure comprising of objects, classes, messages and links between objects, and associations and relationships between classes.

Now wouldn't it be better if our design model were to consist of diagrams that had the same structure as that of the code? Better still, is there a graphical and diagrammatic language that can convert our structure into code? The answer is UML.

There are two types of diagrams supported by UML. One set of diagrams depicts the structure, i.e. the interconnections and resulting topology. This set is also called static diagrams. The other set of diagrams, called dynamic diagrams, depicts the messages that are passed between objects.

### **Example 2.1: College Administration Example – A Case Study**

In this section, we use the college administration system to explain the concepts in object modelling and its implementation in UML. College academic administration revolves around students, teachers and courses. Colleges have departments, and departments offer courses. Students register for the courses offered by a teacher.

We explain object modelling with object models that pertain to college

administration and include actors like students, teachers, HODs, courses, departments, etc. We use this as a running example to learn about OOPS concepts and also while dealing with specifics of C++ and Java in subsequent chapters. Basic to our model is a class called student. It contains information such as roll number, name, etc. We use a class called student and store information about students in a database called StudentMaster. Course files will have the details of students registered for a particular course. All such courses will be a part of a department. Colleges running courses such as BTech, MBA and MCA would in turn be a part of a campus.

## 2.4 Classes and Objects

The object is an entity that contains data and functionality. The college administration has data about students, teachers, and courses on offer. The decision we have to take in OOP design is how to divide data and functionality into meaningful objects that

interact amongst them and produce the result the college administration expects.

The first thing that comes to mind is an object called student. In a college there are several students, but all have the same functionality and individual attributes. Hence, these objects can be grouped into a class called student.

*A class is a collection of a set of objects that share a common behaviour and attributes. Class definition describes attributes as well as functions that implement the behaviour.*

*An object is an entity that contains attributes and member functions and an identity. Objects of common attributes and functionality are grouped together in a class.*

### 2.4.1 Class

*A class is a description or specification for the object, whereas the object is an instance of the class that is created at run-time. There can be one or several instances of the class. In Example 2.2, we show the*

representation of a student class in C++, Java, and UML.

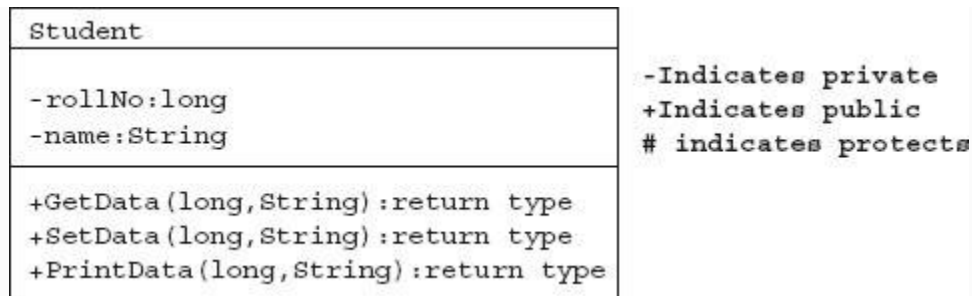
## **Example 2.2: Representation of Class Student in C++**

```
class Student
{ // constructors and member functions
    public:
        Student() {rollNo=0,name="No
Name";}
        Student(int n, char *p)
        {rollNo=n,name=p;}
        int GetRollNo() const { return
rollNo;}
        void SetRollNo ( int n) {
rollNo=n;}
        char * GetName() const { return
name;}
        void SetName( char *p) { name=p;}
    private: int rollNo; char *name; //
member data & pointer to
        name
};
```

## **Example 2.3: Representation of Class Student in Java**

```
public class Student
{private String name;
    private long rollNo;
    public Student( String n , long r )
        {rollNo=r; name=n; }
    public String GetName() { return
name;}
    public long GetRollNo() { return
rollNo;}
} // end of class
```

## **Example 2.4: Representation of Class Student in UML**



**Figure 2.2** Student class representation in UML

### *2.4.2 Notations and Meta Models*

UML employs notations and meta models. A notation is a graphical representation and syntax of the modelling language. A meta model, on the other hand, is a class diagram that is used to define the rigor and intricacies of the language. In the following example, you will see that we will use notations of UML, as well as meta models involving class diagrams, to explain the intricacies of generalization, specialization, etc. Fortunately, we can use the same representations in C++ and UML. A rectangle box represents a class of objects. The name of the class is at the top. In the next part of the box, list the data that each

object contains. In the third part of the class box, list the operations (functions) that an object can carry out.

### *2.4.3 Mandatory Profile*

It defines minimum services, i.e. functionality to be provided by a class. Therefore, it is essential that class definition contain a constructor for the proper initialization of objects. This initialization is achieved through parameterized constructors with default arguments to take care of the correct and required initialization of class attributes and the creation of objects with minimum mandatory services. Consider the example shown below:

---

```
class Student
{ // constructors and member functions
    public:
        Student() { rollNo=0, name="No
Name"; }
        Student(int n, char *p)
        { rollNo=n, name=p; }
        int GetRollNo() const { return
rollNo; }
        void SetRollNo ( int n) {
rollNo=n; }
```



```
        char * GetName() const { return  
name;}  
        void SetName( char *p) {  
name=p;}  
        private: int rollNo; char *name; //  
member data& pointer to name  
};
```

---

If no parameters are provided at the time of the creation of an object, then default values specified by the default constructor are used in initializing the object. The default constructor requires an appropriate behaviour for this class. Hence, we have provided "no name": for the name and 0 for the roll number. Later in the program we can use the SetRollNo() and GetRollNo() functions to modify the initial values:

---

```
Student std; // object created  
std.SetRollNo(6060);  
std.SetName("Gautam");
```

---

In the above declaration, it can be seen that we need to provide two parameters, namely

student's number and student's age. Creating objects without proper initialization often leads to many programming faults and problems. Hence, it should be a good programming practice to define a default constructor as part of a mandatory profile that is put to work by the compiler if the user does not define a suitable constructor. Indeed C++ and Java support the concept of a default constructor as part of a mandatory profile.

#### *2.4.4 Meta Data – Meta Class*

##### **2.4.4.1 Meta Class**

In object-oriented programming, meta class is a class created by language and all other user-defined classes are instances of this meta class. Meta class defines the behaviour of user-defined classes and their instances. Meta class implementation is individual to languages like C++ and Java, and there is no uniform protocol to implement meta class.

When you create an instance of class and class methods to your class, they are actually added to meta class instance. It means that

you are able to change the behaviour of instance that you have created to suit your requirements. In Java, classes you create are instances of `java.lang.Class`. However, Java does not support meta classes directly, but here too there is exactly one instance of class of `java.lang`. We can use the methods defined in `java.lang.Class`: `getName()` or `newInstance()` or `getMethod()`, `getConstructor()`, `isArray()`, `getInterfaces()`. For example, we can get the class description for an object's class by calling `getClass()` on the object.

#### **2.4.4.2 Meta Data**

Meta classes in addition hold meta data, called attributes, i.e. data belonging to meta classes. Meta classes allow meta data to be accessed by classes or packages. The attributes are specified in an object-oriented language as markup script, which meta class is capable of reading and loading at run-time, so that meta data is made available to the programmer.

### *2.4.5 Constraints*

Constraints will help the designer to develop the system as per specifications by ensuring one-to-one mapping of operations.

Specifying operations are usually carried out in graphical modelling so that they define underlying constraints with precision. The net result is a better designed system.

- **Performance:** Performance criteria are defined at the class definition level. Performance dictates the operations. Operations, in turn, decide the responsibilities of the class.
- **Reliability:** Reliability of a class depends on the way the reliability aspect is distributed among the other classes and the way one class interacts with other classes. There should be a match between user operational profiles and functional profiles.
- **Security:** Security issues crop up because the user at operational level is able to exploit some operations that are not part of original the specification. This loophole might have been created inadvertently at operations of other classes' levels. The criteria to be considered to plug these loopholes are described below.
- **Access Control:** Subdivision of functionality must ensure that only certain classes or users are able to initiate and use the functionality with authorization codes.
- **Integrity of Data:** The data is primacy and no unauthorized access can be allowed at submodules. Suitable encryption techniques can solve the problems.

- **Control of User Behaviour:** The normal behaviour of user of a class must be specified and any deviations from this normal and designed process must be recognised by the module, and further exploitation of unintended operations must be denied.

### *2.4.6 Object Creation*

Once a class is defined at run-time we can go ahead and create an instance of the class, i.e. object. Remember that an object is a variable of a class. Therefore, you can define the object just like you define the variable of a data type. In the example that follows, we create an ordinary object called std and 50 instances of class student, and also a pointer that creates an instance of student on the heap memory.

#### **Example 2.5: Creation of an Object of a Class in C++**

---

```
Student Std; // Std is an object of  
class Student  
Student Std[50]; // There are 50
```

```
instances created for Student class  
Std.SetRollNo(8345); // Sets roll no to  
8345
```

---

### **Example 2.6: Creation of an Object of a Class Student in Java and Assigning Initial Values**

```
Student Std = new Student(757,"Anand")//  
std is an object of class Student  
757 is rollNo and Anand is name  
allocated to this instance of object.  
Std is name of the object.
```

---

### **Example 2.7: Creation of an Object of a Class Student in UML and Assigning Initial Values**

### *2.4.7 Garbage Collection*

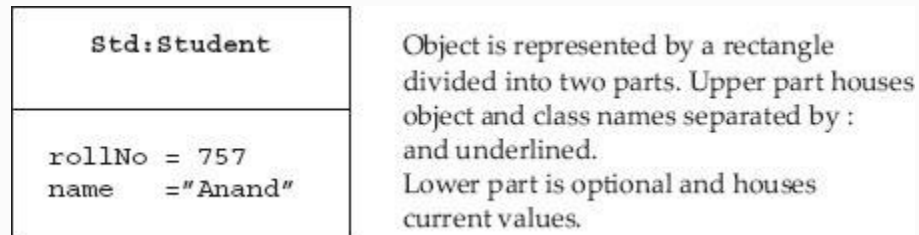
**Garbage collection** is the automatic detection and freeing of memory that is no longer in use. An object becomes available for garbage collection when it is no longer referred to by any other object. Java uses garbage collection rather than explicit destructors found in other OO languages such as C++.

### *2.5 Object Properties*

Object is an entity that has its own data structures and functionality to implement the behaviour. What are the essential properties that an object should possess?

**State:** An instance of an object is an object, created at run-time with initial values set by a default constructor. The values are particular to that instance and can be changed at any time during the life of an object by accessory functions defined within the objects (see Figure 2.3). The values of 757 and Anand are true only at that instance

and hence they represent the state of the object.



**Figure 2.3** An instance of object represented in UML

**Behaviour:** An object's behaviour is decided by functionality defined by the class definition. However, though there can be several instances of the class, i.e. several objects belonging to the same class, all of these instances will have the same behaviour defined by the class. Hence, we have not shown behaviour, i.e. functions separately in [Figure 2.3](#). The behaviour can be in terms of functions and methods that update and modify member data or general functions that serve as interfaces to implement some global functionality. These functions are invoked by other objects and receive the



arguments and provide service to other objects.

**Object name:** When an object is created, it bears a name. For example, Std is the name of the instance of the class called student. This is the identity of the object.

---

```
Student Std; // C++ Std is an object
of class Student
Student Std = new
Student(757, "Anand") // Java
```

---

**Object's identity:** Each instance of the object is different from the other instances of the object, even though they contain the same data values. When an object is created, memory resources are allocated. Therefore, unique id is the memory address of the object.

**Encapsulation:** Objects' member data are declared private and hence the details are hidden and are not available to the outside world. In OO languages like C++ and Java, the definitions for attributes and methods are placed inside a class definition and hence are not available to the outside world,

thus hiding internal details from the outside world. This is known as encapsulation.

## 2.6 Links

We have learnt that object modelling is not simply a collection of isolated objects. As shown in Figure 2.4, the system requires that links between data belonging to different objects need to be defined to achieve some global functionality. There are three distinct objects belonging to three different classes: `batch`, `student` and `Registration`. The class called `batch` will have instances of students registered for all courses for that batch, say mechanical third semester. Here, only the student's number is held and in order to get full details of the student, we need to refer to another object called `Registration`. This means that for each student object, we need to maintain a link to `Registration` object so that we can refer to it for full details of the student. Similarly, we need to maintain a link class

called batch so that we can get batch details of the student.

We will show the implementation of link in C++ and Java and UML as usual. However, as a cautionary note, the examples in C++ and Java require understanding of the rudiments of programming skills in these languages. Hence, first-time readers can skip examples in C++ and Java and concentrate only on UML; they can then revisit these topics after acquiring the necessary C++ or Java skills.

## **Example 2.8: Links in C++**

```
class StudentMaster
{ public: StudentMaster(char *p, int n)
{name=p, rollNo=n;}
    char * GetName() const { return
name;}
    int GetRollNo() const { return
rollNo;}
    private: int rollNo; char *name; //
pointer to name}; // end of
```

```

class
class Student
    { public: Student(StudentMaster *std
){ sm=std;}
    char *GetName() const { return Sm-
>GetName();}
    private: StudentMaster *sm; };
void main()
{StudentMaster *sm = new StudentMaster (
"Anand",757);
    Student * S1 = new Student(sm);
    cout<<s1->GetName();}

```

---

## Example 2.9: Links in Java

---

```

public class StudentMaster
{
    private String name;
    private long rollNo;
    public StudentMaster ( String n
, long r)
    {name=n; rollNo=r; }
    public String GetName() { return
name;}
    public long GetRollNo() { return
rollNo;}
}

```

```
    }// end of class
    public class Student
    { private StudentMaster sm;
      public Student( StudentMaster
std)
        { sm= std; }
    }// end of Student class
```

---

From the above, you can clearly see that whenever an object of student, i.e. sm, is created, we also create a reference to it in StudentMaster. We will show you how these links are created:

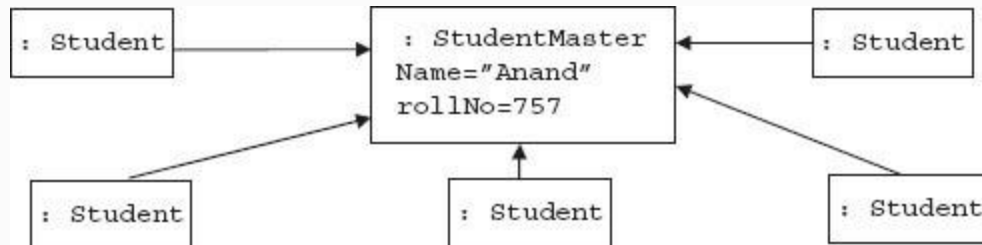
---

```
    StudentMaster sm = new
StudentMaster("Anand",757);
    Student S1 = new Student(sm);
    Student S2 = new Student(sm);
    ..... and so on for
five subjects
    Student S5 = new Student(sm);
```

---

## **Example 2.10: Links in UML**

A link is shown in UML by an arrow pointing to the object:

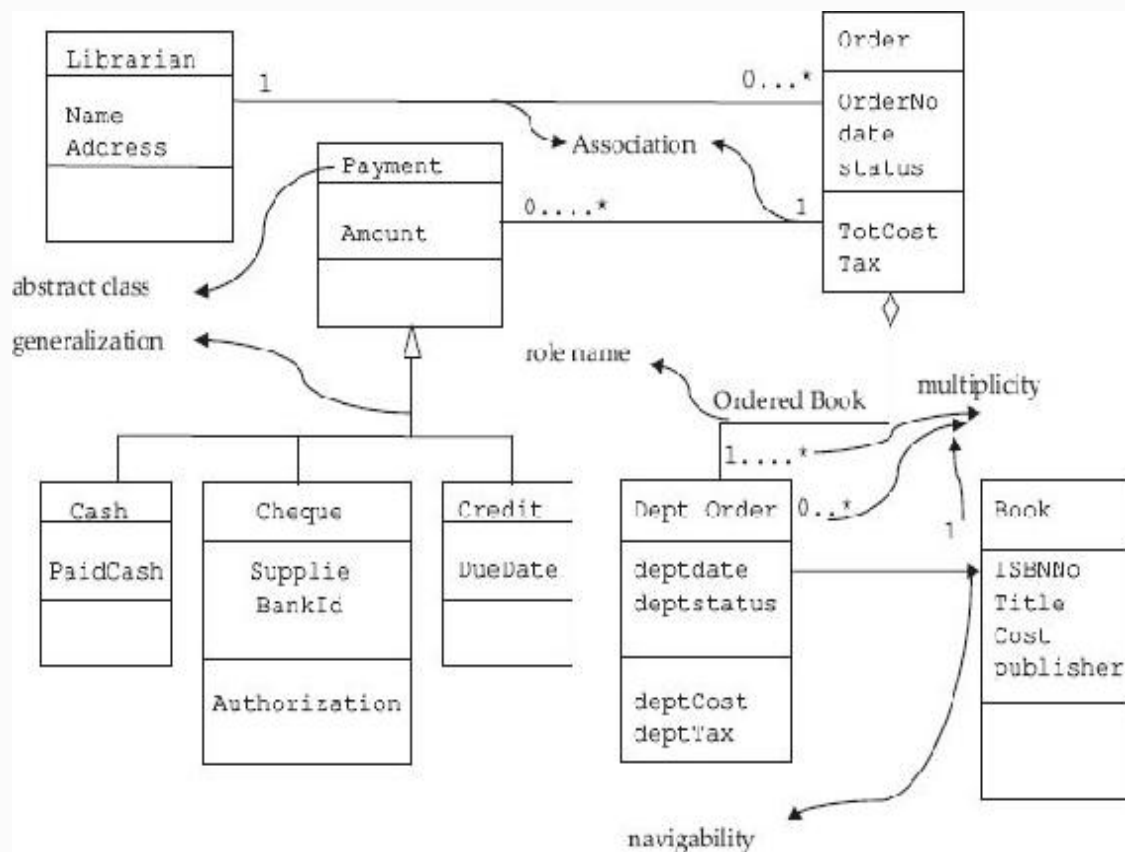


**Figure 2.5** Links in UML

## 2.7 Class Diagrams

Class diagrams provide system overview. They show classes and their relationships with other classes. They are static diagrams. This means that they tell us about the interactions amongst classes, but they do not tell us the response to these interactions. We will explain the class diagrams and various terms involved with the modelling of library book ordering systems. Class diagrams show a librarian making a purchase from a supplier by raising an order and afterwards making a payment. Payment

can be either by cash, cheque or credit. Order, on the other hand, contains book details (see Figure 2.6). A detailed explanation of the above class diagrams and concepts are explained in the succeeding sections.



**Figure 2.6** Class diagram for library procurement system in a college

## 2.8 Class Hierarchy

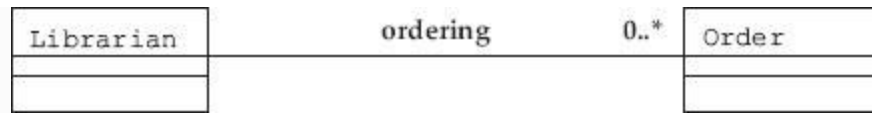
In order to bring out relationships and dependencies amongst classes, we need to arrange classes into hierarchies.

### *2.8.1 Associations*

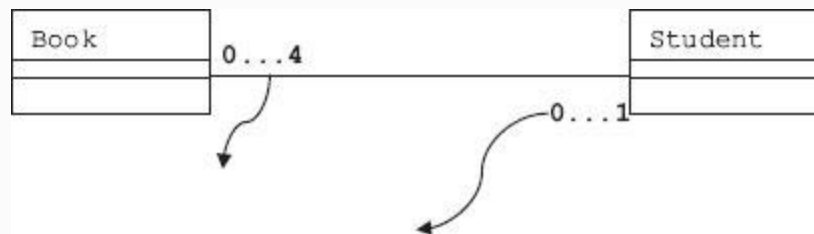
**Associations are a type of hierarchy that show** relationships between classes that are independent and are of equal standing. These relationships are shown by lines between classes; they are usually labelled with the name of the association. Classes in an association usually occupy equal places within a hierarchy. For example, in Figure 2.6, librarian and order are independent classes.

**Multiplicity:** 0..\* at order class means that librarian can place an unlimited number of orders, i.e. from 0 to unspecified number of orders. 0...1 means 0 to 1. Similarly, we can also state the maximum number of books that can be borrowed by a student as 0...4, as shown in Figure 2.8.





**Figure 2.7** Association between classes. Association is named : ordering



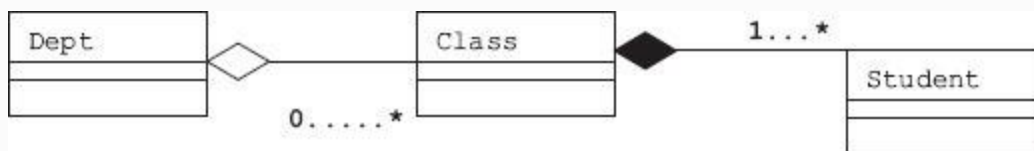
**Figure 2.8** Association 0...1

A student can borrow 0 to many books.  
 A book can be borrowed by at most 1 student (see [Figure 2.6](#)). The navigability shown above implies the direction of the association and who owns the association's implementation. The navigability tells us that we can find out details of books by querying `DeptOrder`. Also make a note that `DeptOrder` is the owner of the association.

## 2.8.2 Hierarchies with Interdependent Classes

There are two types of hierarchies:

- **Whole/part of hierarchy.** In this type of hierarchy, one class is included in the other class. This relationship is called **has a** relationship. Further, there are two subtypes in a whole/part hierarchy. They are as follows:
- **Aggregation.** In this type, objects are not fixed. They can be included or removed. For example, consider the example in [Figure 2.9](#). A department has some classes that have some students. Aggregation means “**part of**” relation.



**Figure 2.9** Aggregation relationship. Student is a part of class, i.e. part of a department

In a class hierarchy involving whole/part, parts are essential, like in the case of a book. A book cannot exist without pages, a passenger train cannot exist without compartments, etc. In [Figure 2.9](#), a class

cannot exist without students. Hence, it is shown as a black diamond. A class contains 1 to many students, whereas a department can have no class, i.e. empty department. Hence, it is shown as a white diamond.

- **Composition.** This is another type of whole/part type of hierarchy. In this type, the object is a whole or integral part of the whole. Examples are human has a heart; a computer has a microprocessor in it. Composition means “has a” relationship, i.e. the whole object contains a part.

Aggregation is shown by a line with a **hollow diamond** pointing to the whole class, while composition uses a **solid diamond** instead. One way to implement the whole/part hierarchy is to simply declare an instance of the aggregate object as an attribute, while defining the container class:

### **Example 2.11: Example of Container Class in C++ and Java**


---

```
class Date
{public:
    Date ( int d=0, int m=0,int y=0)
:dd(d),mm(m),yy(y) { } //
constructor
    void SetDate( int d, int m, int y){
dd=d,mm=m,yy=y;}
    int GetDD() const { return dd;};
int GetMM() const { return
mm;};
    int GetYY() const { return yy;};
private: int dd; int mm; int yy;
};
class Student
{ public:
    Student();
    ~Student(){}; // Default destructor
    int GetRollNo() const { return
rollNo;}
    void SetRollNo ( int n) { rollNo=n;}
    Date GetDOB() const { return
dateDOB;}
    Date GetDOJ() const { return
dateDOJ;}
    void SetDOB( Date dtb) {
dateDOB=dtb;}
    void SetDOJ( Date dtj) {
dateDOJ=dtj;}
private: int rollNo; Date dateDOB;
Date dateDOJ; };
```

---

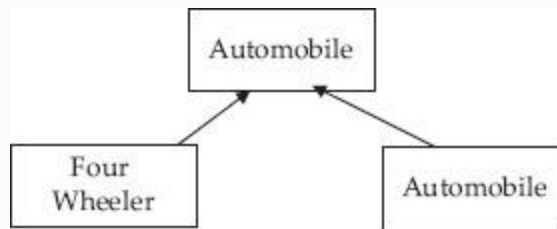
In the above example, observe that we have defined a class called `date`, with its own data members and member functions. The `date` class has `dd`, `mm`, `yy` as private attributes. Now a class called `student` has declared instances of `date` for date of birth and date of joining as `dateDOB` and `dateDOJ`.

### *2.8.3 Hierarchies with Independent Classes*

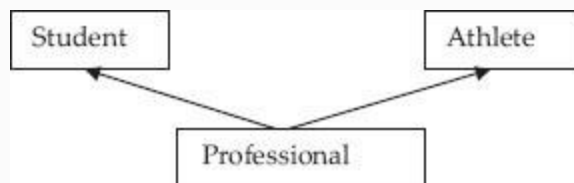
The classes under this category have an *is a* relationship and are also referred to as **generalization/specialization**. In OOPS programming language, the implementing mechanism for this feature is called inheritance. For example, we have modelled Human Beings hierarchy in [Figure 1.5](#). UML uses open arrow head  to show inheritance relationship.

We have shown an inheritance relationship in [Figure 2.10](#) where in Four wheeler and two wheeler inherit from an Automobile. A base class is known as **Super** class in the literature. We will use base class in C++ and Super while dealing with java &

UML. The derived class can extend the Super by adding new properties, and by selectively **overriding** existing properties of the super class. For example  
Father (Super) Moves from place A to B by Road, while the Son ( sub class)  
Moves from place A to B by Air.



**Figure 2.10** Inheritance hierarchy



**Figure 2.11** Multiple inheritance – more than one super class

If a class derives from more than one superclass, it is called multiple inheritance. Java does not support multiple inheritances, while C++ supports it. There can be confusion and ambiguity regarding inheriting attributes and qualities from more than one parent class. Java solves this problem of ambiguity by introducing a concept called interface and C++ by resorting to explicit function calls. In Java's interface, class will use a Java's specification called **implements** that will allow implementing class to implement the code required by an interface. You will learn more about this aspect in language specifics.

### **Example 2.12: Example of Multiple Inheritance in C++**

```
class Student
{ public:
    ..... class declaration here.....
    private:
```

```

};
class Athlet
{ public:
    ..... class declaration here.: };
class Professional : public Student,
public Athlet
{ public:
    ..... class declaration here.
.....
private:
};

```

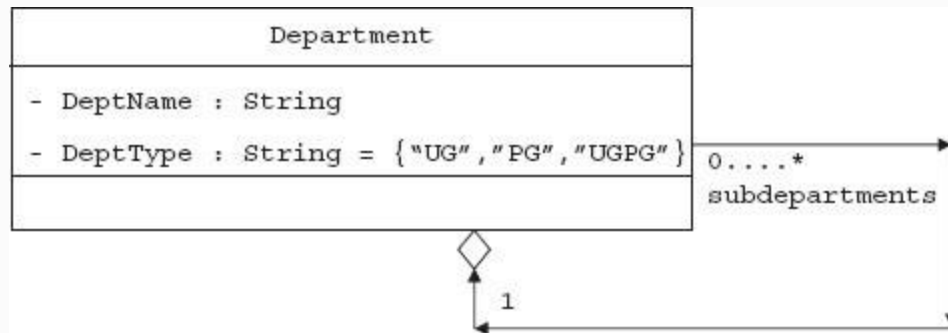
---

## 2.9 Object Diagrams

Object diagrams show instances of class diagrams. For this example consider the college department organization. A class can have several departments under it. A department in an engineering college can have many departments under it. Figure 2.12a shows a class diagram called department. A department can also contain many subdepartments. For example, a department like electrical engineering can have subdepartments like electronics department and communications department. We can say that a department is

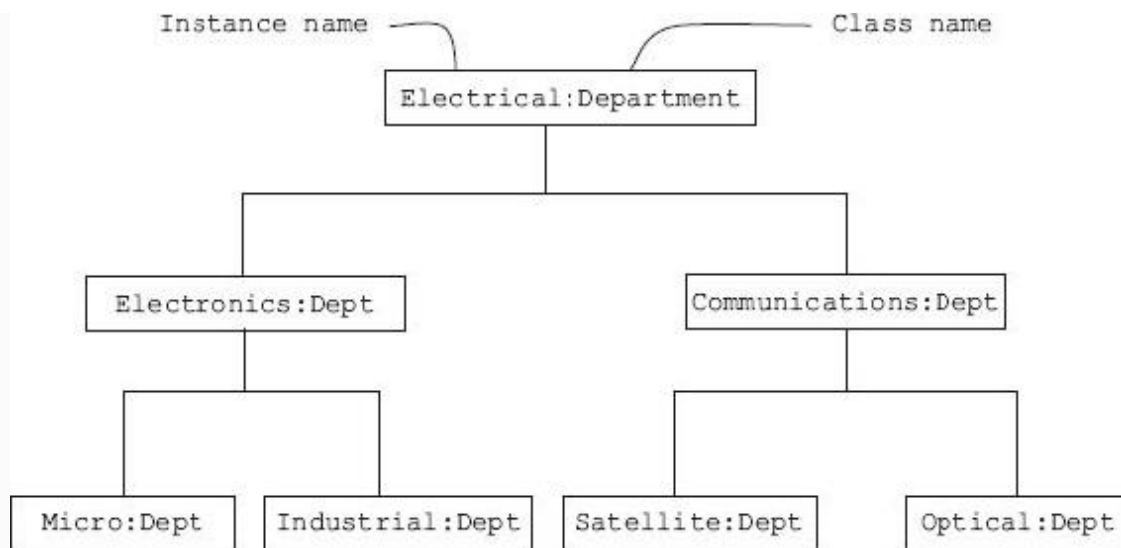


a collection of subdepartments. This in UML is a representation of aggregation and is symbolized by placing a diamond.



**Figure 2.12a** Department contains 0 to many sub departments – Aggregation

The self-loop shown in Figure 2.12a depicts aggregation. It denotes that department can contain 0 to \*, means 0 to many subdepartments. These departments are contained in 1 instance of the department. Hence, observe the diamond shown to indicate aggregation. The object diagram shown in Figure 2.12b instantiates the class diagram, replacing it by a concrete example.



**Figure 2.12b** Object diagram showing department containing several subdepartments

## 2.10 Communications and Message Passing

Executing a program in today's technology is no more the execution of a sequence of logical programming steps. Most of the GUI programs have to respond to signals from external units such as mouse, sensors, etc. But the question is who will monitor trigger from external units? Is it the program written by the user or the operating system? The answer is an OS monitors the signals from external units and sends a signal to our

program. Our program will have an interface to receive and interpret and take necessary actions on the signals sent by the OS. Indeed entire Windows and other modern operating systems achieve their tasks through this message passing mechanism.

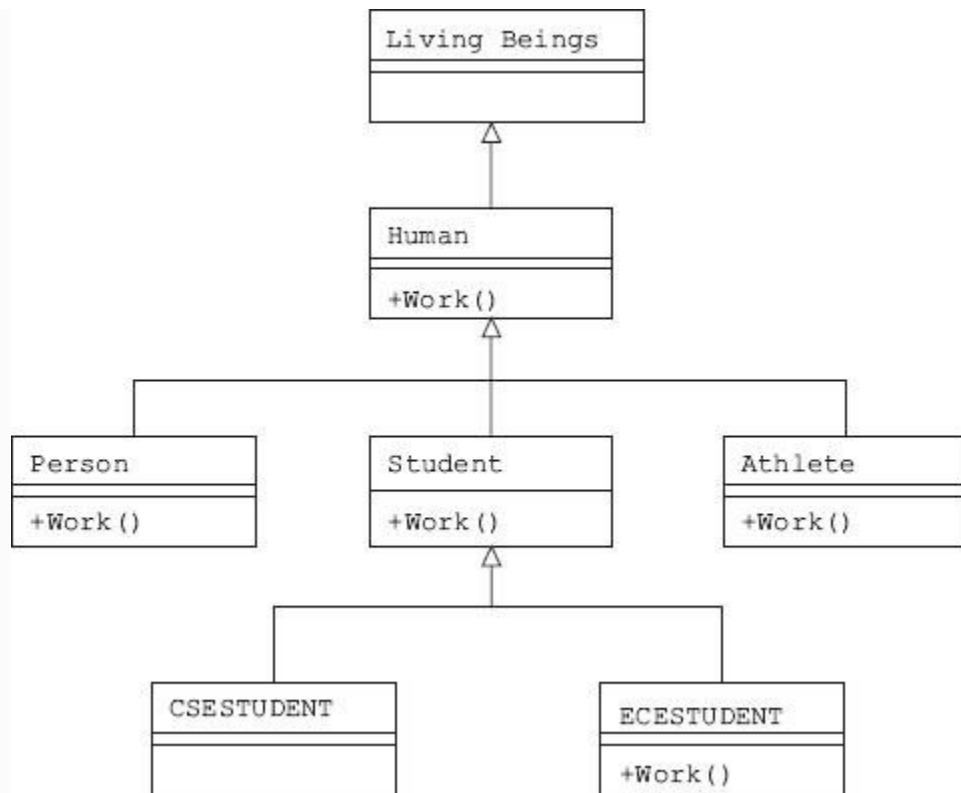
There are also other sets of communications by cooperating objects. Objects send and receive messages. Sending a message is accomplished by invoking a function belonging to another object and passing parameters. Receiving a message would imply that a function belonging to the current object has been invoked by another object. Objects can thus respond to messages sent by OS as well as other objects.

## 2.11 Polymorphism

We have understood that in inheritance relationship subclass derives from superclass both methods and attributes. In inheritance, the subclass can extend and simply use the functionality of the base class or use its own functionality with totally different implementations of the same

function. We have called this feature overriding base class functions.

We have shown in Figure 2.13 a human hierarchy. All humans have to work, so the human class defines a function called `Work()`. All Humans `Work()`, but a specialized class like student, which is a subclass of human, must work in a college and in libraries and laboratories. Hence, you will see that student has overridden the base class function called `Work()` with its own specialized `Work()` method. Also observe that CSESTUDENT would like use (inherit) the same `Work()` as that of its base class student. Hence, he has not defined its own version and hence base class function `Work()` is available to it. On the other hand, ECESTUDENT probably has to work in hardware laboratories and hence has overridden base class `Students Work()` with its own implementation.



**Figure 2.13** Polymorphism at work: ECE students has own work(): CSE student works with Student Work() because CSESTUDENT has not overridden.

When there are several subclasses implementing a function, it is critical that the correct function of the correct derived class be invoked. That is, at run-time, the correct function has to be invoked through a function call. The feature that ensures this correct run-time binding of function is

called **dynamic binding**. Polymorphism means to ensure that objects of base class behave correctly and as planned. For example, CSESTUDENT must use `Students Work()` and ECESTUDENT must use its own `Work()` and so on.

## 2.12 Abstract Class

While designing an OO system, we may need a base class with no function implementation at all. The base class will have only the names of the functions and no implementation of these functions.

However, the derived classes from this base class will override the base class function in which each of the overridden function will have its own implementation.

Polymorphism and the feature of dynamic binding would ensure that depending on the user's requirement, the correct method belonging to the appropriate object would be linked to a function's call. The interface feature of Java is an abstract class. In C++, a base class with pure virtual functions can be considered as an abstract class. An abstract

class, since it has no implemented functions, cannot have an object instantiated.

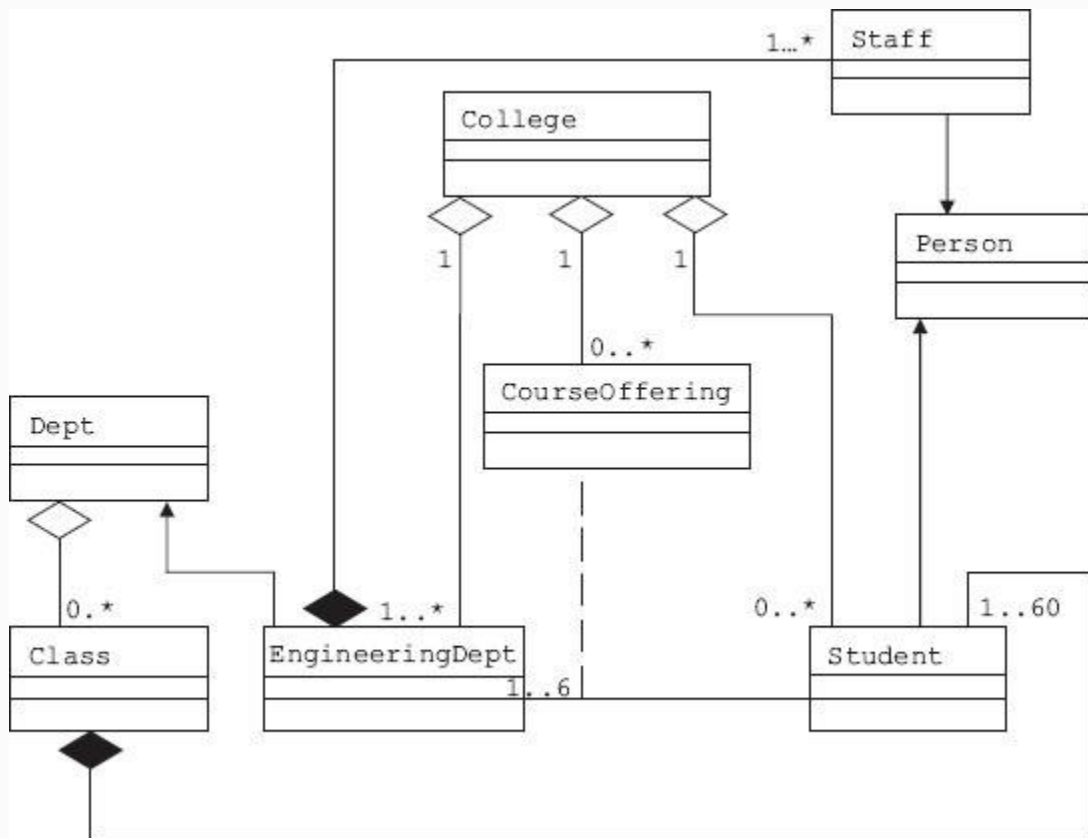
## 2.13 Concrete Class

As opposed to abstract classes, the concrete class implements all the functions and methods declared in it. Hence, a concrete class is a class that can have an object or instance of object instantiated. In Figure 2.13, the student class is a concrete class.

### *2.13.1 OOPS – A Case Study in Object Modelling*

We will carry out a detailed modelling exercise to fortify what we have learnt about relationships. We will study the college administration system as it pertains to college, departments, and students and relationships, as shown in Figure 2.14. There are two main classes: Person and Dept. These two are superclasses (base classes) and hold attributes common to all persons and departments such as the name of a person or department, college code, and all other relevant common data. Student and

staff are subclasses derived from super person and is a special kind of person. It contains specialized attributes such as identification number, address, dept code and other relevant data. This relationship is inheritance and is shown with the symbol.



**Figure 2.14** Relationships between college and students



**Aggregation** is shown by a line with a **hollow diamond** pointing to the whole class. A department has some classes that have some students. Aggregation means “**part of**” relation and is shown as ◇.

**Composition** uses a **solid diamond** instead. In this type, object is a whole or integral part of the whole. Composition means “has a” relationship, i.e. the whole object contains a part and is shown in UML as ◆ In Figure 2.14, the class has 0...60 students. The class is composed of students. Any method a class needs to manipulate students will be implemented in class, and these methods are available to any derived class(sub class) of a class.

You can observe that `EngineeringDept` is derived from `Super Dept` and is a specialized department. It will have additional information peculiar to an engineering department such as specialization, resource persons, and link to students belonging to the department. Look at the `CourseOffering` class. It implements the course offering relationship

that exists between a student and an EngineeringDept. Also observe that CourseOffering is ***also a part of*** college which is a list of all engineering departments offering courses. A college needs some other objects to keep track of courses that have been offered. Further class called CourseOffering can have additional functionality such as tracking which course a student has been offered. The class CourseOffering can also track courses that are currently running and/or courses that have concluded. Make a note that we have defined a class called CourseOffering to manage the association relationship between EngineeringDept and student. The dashed line ----- in UML indicates the relation just described. The above association shown in Figure 2.14 means that a student can be offered 1..6 courses by an EngineeringDept and these courses are tracked and monitored by CourseOffering class. The relationship between college and EngineeringDept is

aggregation. We can say that  
EngineeringDept is ***part of*** college.

## 2.14 Summary

1. Module: The basic building block in an OO language is a module. Modules can be defined as logical collections of classes and objects. A module is not a function or a procedure or an algorithm, but rather the classes and objects and messages passing between them.
2. Object orientation is a technique of modelling a real-world system in software based on objects.
3. OO programming is a collection of cooperating objects which are in turn instances of a class. The cooperating classes have a hierarchy of relationships defined by inheritance relationship.
4. OOP language means: objects, classes, inheritance.
5. Class: A class is a collection of a set of objects that share a common behaviour and attributes. Class definition describes attributes as well as functions that implement the behaviour.
6. Mandatory profile: This defines minimum services, i.e. functionality to be provided by a class.
7. **Object** is an entity that contains attributes and member functions and an identity. Objects of common attributes and functionality are grouped together in a class. An object's behaviour is decided by functionality defined by the class definition.
8. The unique identification for an object is the memory address of that object.
9. **Encapsulation** is the process of hiding all the internal details of an object from the outside world through

- access specifiers such as private and protected.
10. Link connects data belonging to different objects to achieve some global functionality. A link is shown in UML by an arrow pointing to the object. A link is also an instance of an association.
  11. Class diagrams provide system overview. They show classes and their relationships with other classes. They are static diagrams. This means that they tell us about the interactions amongst classes, but they do not tell us the responses to these interactions.
  12. Hierarchy: An ordering of classes. The most common OO hierarchies are inheritance and aggregation.
  13. Association is a type of hierarchy that shows relationships between classes that are independent and are of equal standing. An association is a relationship between two or more classes.
  14. Multiplicity: An attribute that quantifies an association between objects.
  15. Whole/part of hierarchy: In this type of hierarchy one class is included in the other classes. This relationship is called **has a** relationship.
  16. Aggregation means “part of” relation. An aggregate object has other objects. Other objects are part of aggregate objects. Aggregation is shown by a line with a **hollow diamond** pointing to the whole class ◇.
  17. Composition: This is a special case of aggregation where whole cannot exist without parts. Composition uses a solid diamond ◆.
  18. Hierarchies with independent classes: The classes under this category have an is a relationship and are also referred to as **generalization/specialization**.
  19. Inheritance specifies **is** type of relation. A base class is known as **superclass**. A derived class is also known as a subclass. Inheritance is shown in UML with the symbol ▷.

20. Derived subclass not only inherits the properties of the superclass, but it can also extend and modify its behaviours and attributes.
21. Object diagram shows instances of class diagrams.
22. Dynamic binding: When there are several subclasses implementing a function, it is critical that the correct function of the correct derived class be invoked. That is, at run-time, the correct function has to be invoked through a function call. The feature that ensures this correct run-time binding of function is called **dynamic binding**.
23. Polymorphism means to ensure that objects of a base class behave correctly and as planned.
24. Abstract class: This is a base class with no function implementation at all. However, the derived classes from this base class will override the base class function in which each of the overridden functions will have its own implementations. The interface feature of Java is an abstract class. In C++, a base class with pure virtual functions can be considered as an abstract class.
25. An abstract class, since it has no implemented functions, cannot have an object instantiated.
26. Concrete class is a class that can have an object or instance of object instantiated.

## Exercise Questions

### Objective Questions

1. Which of the following statements are true regarding OO language?
  1. OO language comprises objects, classes and inheritance.
  2. Modules are logical collections of classes and objects.
  3. Modules are a collection of functions.

4. Class definition describes attributes as well as functions.

1. i
2. i and ii
3. i, ii and iv
4. i, ii and iv

2. Which of the following statements are true regarding objects and classes?

1. Objects contain member functions and attributes.
2. Objects contain member functions and attributes and an identity.
3. The unique identification for an object is the memory address of the object.
4. Mandatory profiles define maximum service provided by the class.

1. i
2. i and ii
3. ii and iv
4. i, ii and iv

3. Which of the following statements are true regarding class diagrams?

1. Class diagrams provide system overview.
2. Class diagrams indicate the interactions amongst classes.
3. Class diagrams indicate the responses to interactions.
4. Class diagrams show classes and their relationships with other classes.

1. i
2. i and ii
3. ii and iv
4. i, ii and iv

4. Which of the following statements are true regarding object methodology?

1. A link connects data belonging to different objects.
2. A link is shown in UML by an arrow to a class.
3. A link is also an instance of an association.
4. Association shows relationships between classes that are independent and are of equal standing.

1. i, iii and iv
2. i, ii and iii
3. ii and iv

4. i, ii and iv

5. Which of the following statements are true regarding association?

1. An association is a relationship between two or more classes.
2. Multiplicity is an attribute that quantifies an association between objects.
3. Whole/part of hierarchy means “is” type of hierarchy.
4. Aggregation means “part of” relation.

1. i, iii and iv
2. i, ii and iii
3. ii and iv
4. i, ii and iv

6. Which of the following statements are true regarding aggregation?

1. Aggregation means “whole of” relation.
2. Other objects are part of aggregate objects.
3. Aggregation is shown by a line with a **hollow** diamond pointing to the whole class.
4. Aggregation means “is” relation.


1. i, iii and iv
2. i, ii and iii
3. ii and iii
4. i, ii and iv

7. Which of the following statements are true regarding composition?

1. Composition means whole cannot exist without parts.
2. Composition uses a solid diamond.
3. Composition is shown by a line with a hollow diamond pointing to the whole class.
4. Composition means “is” relation.

1. i and ii
2. i, ii and iii
3. ii and iii
4. i, ii and iv

8. Which of the following statements are true regarding inheritance?

1. Inheritance means “is” type of relationship.
2. Inheritance means a hierarchy with an independent class.
3. A base class is known as superclass.
4. Inheritance is shown in UML with symbol 

1. i and ii
2. i, ii, iii and iv
3. ii and iii
4. i, ii and iv

9. Which of the following statements are true regarding object-oriented methodology?

1. Dynamic binding ensures correct run-time binding of classes.
2. Dynamic binding ensures correct run-time binding of functions.
3. Polymorphism ensures correct behaviour of base classes.
4. Polymorphism ensures correct behaviour of derived classes.

1. i and ii
2. i, ii, iii and iv
3. ii and iv
4. i, ii and iv

10. Which of the following statements are true regarding object-oriented methodology?

1. An abstract class can have an object instantiated.
2. Concrete class is a class that can have an object or instance of object instantiated.
3. An abstract class is a base class with no function implementation at all.
4. In C++, a base class with pure virtual functions can be considered as an abstract class.

1. i and ii
2. i, ii, iii and iv
3. ii, iii and iv
4. i, ii and iv

**Short-answer Questions**

11. Define object-oriented programming.
12. Define class and object.
13. What is garbage collection as it applies to object-oriented methodology?
14. What is an abstract class?



15. What is a concrete class?
16. Explain aggregation.
17. Explain composition.
18. Explain multiplicity of an association with examples.
19. Distinguish class diagrams and object diagrams.

#### **Long-answer Questions**

20. Explain object modelling.
21. What are mandatory profile, meta model and meta data?
22. Why are specifying constraints important?
23. How are objects created in C++ and Java? State the properties of objects.
24. What are links in object modelling? Give examples of link implementations in Java, C++, and UML.
25. Explain the class diagram with an example and explain the various features involved.
26. Explain the object diagram with an example and explain the various features involved.
27. Explain class hierarchies with examples.
28. Explain the polymorphism with an example and explain the various features involved.

#### **Assignment Questions**

29. Develop a class diagram for the inventory control department of an industry.
30. Develop a class diagram catalogue display for a large departmental store wherein customers look into models available and place orders online using credit cards.
31. Carry out object modelling for an automatic teller machine of a bank.
32. Draw a class diagram for a fire alarm system. Assume that the central monitor monitors fire alarms and when fire incidents occur, informs the fire station.

### **Solutions to Objective Questions**

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

# 3

## Extensibility and Reusability – Inheritance at Work

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to understand the concepts of OOP paradigm regarding*

- Extending and deriving of new classes from existing classes.
- Concepts of containment and inheritances.
- Overloading and overriding base class functions.
- Single and multiple inheritances.
- Virtual functions.
- Interfaces provided by Java language.
- Run-time polymorphism and data binding and class as ADT.
- OOP concepts like RTTI, decoupling and STL.

### 3.1 Extendibility and Reusability of OOP Paradigm

Classes have relationships amongst them which are responsible for making the OOP languages versatile and powerful. There are several tools and features provided by OOP languages to ensure reusability and extendibility, as follows: composition / containment, generalization / specialization in inheritance, virtual functions and abstract data types (ADTs), and standard template libraries (STL).

### 3.2 Extending / Deriving New Classes

We can derive a new class based on an existing class. This is possible through the inheritance and containment property afforded by OOP languages. Containment is class within a class, i.e. containment ensures all the member functions and member data of a contained class to the class containing the class.

Inheritance, on the other hand, is a technique to create a new class from an existing class by adding additional and many

a times specialized functionality to it. Inheritance provides access to all non-private (public and protected) member data and member functions to the descendant class.

### *3.2.1 Containment: Class within a Class – Container Class*

Container class is one of the techniques provided by OOP languages to achieve reusability of the code. Container class means a class containing another class or more than one class. For example, a computer **has** a microprocessor in it. In other words, it can also be called **composition or aggregation**.

In composition, one class (let us say Host) stores an instance of another class (let us call it Guest class) in an instance property. The Host class delegates work to the Guest class by invoking methods on that instance.

As an example, a Student class can have a data member belonging to a string class. Then we would say that the string class is contained in the Student class. The

advantage of containment is that we can use a string class within a Student class to store details such as name and address belonging to a string class. Similarly, if we define a class called Date with day, month and year information, we can define the object of Date within a class called Student and define Date-related data such as DOB, DOJ, etc.

**Composition is a “has” type of relation.**

### **Example 3.1: Composition / Containment of C++**

Composition means that the object comprises of several subobjects. For example, BankAccount has CurrentAccount and SavingsAccount. The containment relationship by drawing a line with a black diamond from the containing object to the contained object is shown in Figure 3.1, as per UML specifications.



**Figure 3.1** Containment – composition

## Implementation in C++

---

```
class BankAccount { ... };
class SavingAccount
{private: BankAccount Account;
};
```

---

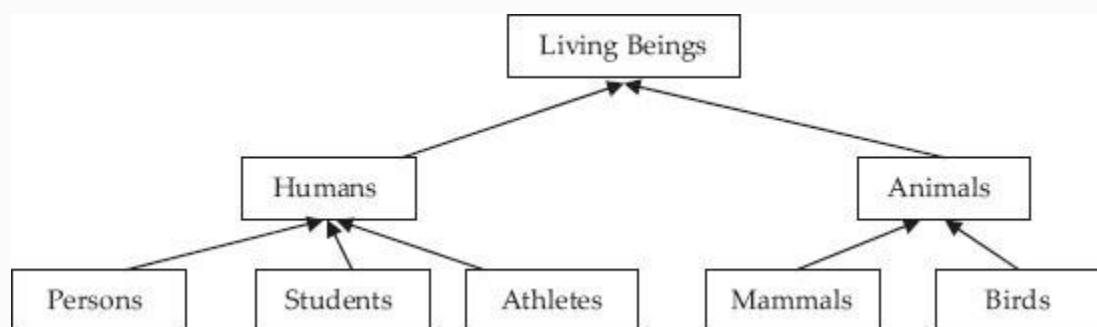
## Implementation in Java

Not supported in Java. The lifetime of a composite object depends completely on the garbage collector.

### *3.2.2 Inheritance and Class Hierarchy*

Reusability of code is one of the strong promises made by OOP languages like C++ and Java, and inheritance is the tool selected to fulfill this promise. The concept of inheritance is not new to us. We inherit property, goodwill and name from our parents. Similarly, our descendants will

derive these qualities from us. Study the inheritance class hierarchy shown in [Figure 3.2](#). Humans and Animals derive qualities from living beings. Professionals, Students and Athletes are derived from Humans. We say these three categories have inherited from Human. Class Human is called a base class and Student and Athlete are called **derived class or subclass**. In Java language, base class is referred to as **superclass**. What can be inherited? Both member functions and member data can be inherited. Note that the direction of the arrow to indicate the inheritance relation is pointed upwards as per modelling language specifications.



**Figure 3.2** Inheritance hierarchy



Inheritance specifies **is** type of relation. Observe that a student is a human. Similarly, a mammal **is** an animal. Human is a **base class / superclass** and Student is a derived class / subclass. Derivation from base class is the technique to implement **is** type of relation. A base class can have more than one derived class.

When do we use inheritance? You inherit so that you can derive all the functionality and member data from base class, and derived class can add its own specialized or individualistic functionality. We can therefore say that subclasses are more specialized versions of their base class or superclass. They can derive all the functionality of superclass and also introduce its own specialized behaviour.

For example, Athlete derives all functionality and attributes of Human and in addition adds sports and athletic functionality and attributes on its own. Let us say that base class Human as defined a method called `Work()` and an attribute like

hoursAtWork to indicate the number of hours Human spend at work. All the three subclasses of Human, namely, Person, Student and Athlete, will derive the functionality and attribute defined by the superclass. Thus, the code need not be written three times – writing it once is sufficient. In the example that follows we will show you sample statements for inheritance and other related topics in C++ and Java. However, the user is advised to refer to relevant chapters for detailed treatment of the topics.

### **Example 3.2: Inheritance and Derivation in C++**

```
class Human
{public:
    protected: // variables declared as
protected are public to
                derived classes
                // protected variables
```

```
};  
    // Student inherited from Human,  
    inheritance type is public  
class Student : public Human  
{public:  
    private:  
};
```

---

Note that members declared as private are not available to derived class. If you declare these member data as public then security is compromised. So what is the way out? Declare the member data as protected as shown above. If a public member function has the object, i.e. the object has invoked the member function, then that function can access

- All the public member functions and public member data.
- Public functions of a class can access all the private member data and member functions of its own class and protected member data and member functions of their base classes.

So, if you want the member data to be passed to derived class in inheritance, declare them as **protected**.

---

## Example 3.3: Inheritance and Derivation in Java

In Java, inheritance is achieved by extension. Suppose that we have a base class called Human with attributes and methods. Then the derived class, called subclass, is created by using the key word **extends**. **This is shown in the example below:**

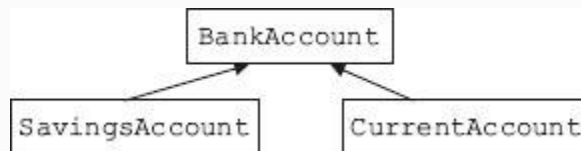
```
Class Human
{ // member data & methods }
class Student extends Human {
// new fields and methods defining a
Student would go here}
```

This gives Student all the same fields and methods as Human, yet allows its code to focus exclusively on the features that make it unique.

### 3.2.3 Generalization vs Specialization

Generalization describes the relationship whereas inheritance is a tool or method of

OOPS languages to implement generalization. We can say that generalization is a generic name for inheritance. Figure 3.3 shows an example.



**Figure 3.3** Generalization – generic name for inheritance

Generalization means that the derived subclass is a subtype of base object. For example, CurrentAccount is a BankAccount . Therefore, we can say that BankAccount generalizes the common behaviour and attributes of CurrentAccount and SavingsAccount . Common functionality be provided only in base class. That is, base class is generalization and derived class or subclass is specialization.

## Example 3.4: Generalization Implementation in C++ and Java

### Implementation in C++

---

```
class BankAccount { ... };  
class SavingAccount : public BankAccount  
{ ... };
```

---

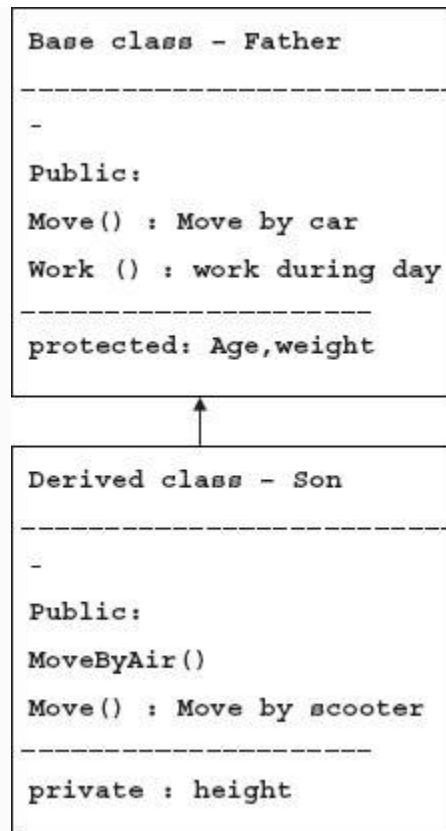
### Implementation in Java

---

```
public class BankAccount { ... }  
public class SavingAccount extends  
BankAccount { ... }
```

---

Now let us discuss the specifics of inheritance. Consider the base class called Father and derived class called Son, as shown in [Figure 3.4](#). As a descendant you are privileged to inherit all non-private member data and member functions.



**Figure 3.4** Inheritance at work

### 3.3 Overriding Base Class Functions

Suppose your father has a car. You can inherit and use the car for your mobility or you can buy a scooter and use the scooter instead. That is, you have **overridden** your father's method of `Move()` with your own functionality. So when you call `Move()`

method, it calls your own move method and you will have to move on a scooter. If you want to use the car, you should call for `Move()` function defined by Father class explicitly like `Father::Move()`.

Suppose base class (super) defines attributes such as age and weight, you can add additional attributes like height. Let us also suppose that super defines functionality such as `Move()` and `Work()`. Suppose you need a functionality to move by air, which your parent did not define, you can define your own function `MoveByAir()`.

### 3.4 Overloading

A class can declare several variations in a function with the same name but with different implementations. This is termed as overloading. We show below base class overloading its `Move()` method:

---

```
Move() { cout<<"move by car":  
Move(int m) { cout<<" move by Bus"); //  
m =1  
MoveFamily(int m , int n ) { cout<<"move  
with family"); // n=number
```



---

Now if derived class Son decides to override only one of the Move() function with its own implementation like: Move() { cout<<"Move by scooter"}).

Then, balance two overloaded functions of base class shown below will be truncated and will not be available to derived class, namely:

---

```
Move(int m) { cout<<" move by Bus"); //  
m =1  
MoveFamily(int m , int n ) { cout<<"move  
with family"); // n=number
```

---

However, subclass can call these functions by explicit mention of base class like  
Father:Move(int m) .

### 3.5 Single and Multiple Inheritances

Inheritance means the ability to derive a new descendant class with additional functionality from base class. Inheritance is a powerful tool in the hand of programmer to define a new class from existing classes.

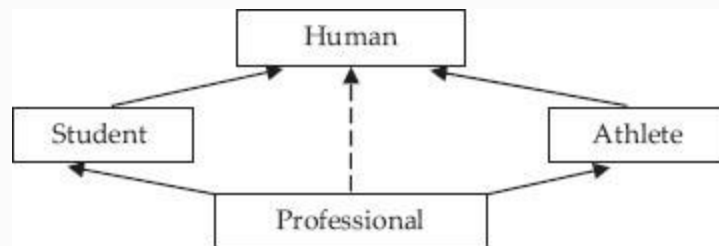
There are different types of inheritances supported by OOPS languages.

In Figure 1.6a, we have shown **single inheritance**, wherein the derived class Person inherits from base class Human .In Figure 1.6b, **multilevel inheritance** is depicted, wherein Student inherits properties from Person, which in turn inherits from base class Human. In Figure 1.6c, we have shown **hierarchical inheritance** involving one base class and three derived classes.

Figure 1.6d depicts **multiple inheritance**, where base class is called virtual because single base class is being used by two derived classes, namely, derived class 1 (DC1) and derived class 2 (DC2). Further, you can see that derived class 3 (DC3) inherits from both DC1 and DC2. This type of inheritance is also called hybrid inheritance as there is a single derived class DC3 from two base classes DC1 and DC2.

### 3.6 Virtual Inheritance

Look at Figure 3.5. There is only one base class for both DC1, student, and DC2, athlete. The base class is called virtual base class and inheritance is termed as virtual inheritance. Virtual inheritance ensures that ambiguity that otherwise exists in case of more than one base class, does not exist.



**Figure 3.5** Hybrid inheritance and virtual Inheritance

In some situations, single inheritance is not sufficient and multiple inheritance leads to ambiguity. Consider the example presented in Figure 3.5.

A Student works at college laboratories and an Athlete works out at the gym after college hours. But let us say that recruiting companies need a Professional who works at college laboratories and also takes keen

interest in gym and sports. We need a professional who inherits `Work()` of Students and also `Work()` of Athletes on requirement. In addition, the Professional must retain all the basic qualities of Human (shown as dotted arrow). Hence the relationship can be termed as hybrid inheritance.

## **Multiple Inheritance is Deriving a Class from More Than One Base Class**

Also notice in Figure 3.5 that both Student and Athlete have the same base class, Human. This type of base class is **called virtual base class** because, though there is only one base class common to both derived classes, each of the derived class feels it has its own base class. In order to facilitate inheritance of properties, functions and member data selectively from Student and Athlete, we need to declare these two classes as **virtual inheritance from the base class**. The relationship is called **virtual inheritance** because, though there are base classes called Student and Athlete for

derived class Professional, they act as if they do not exist and they are there only to pass whatever functions are demanded by derived class Professional.

We would declare virtual inheritance as shown below:

---

```
class Student:virtual public Humans
class Athlet:virtual public Humans
```

---

Professional class is declared as shown below

---

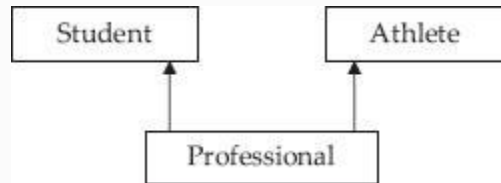
```
class Professional : public Student,
public Athlet
```

---

### 3.7 Problems with Multiple Inheritance

C++ no doubt provides a powerful feature such as multiple inheritance. However, this feature leads to ambiguity as regards deriving from more than one base class. In the example shown in [Figure 3.6](#), professional inherits both from Student and Athlete. Imagine a case where both Student and Athlete classes define a function called

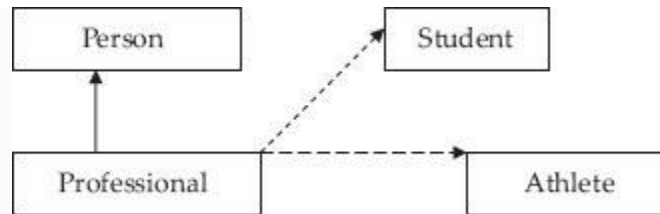
`Work()` . Which function will professional then inherit? Surely ambiguity exists here.



**Figure 3.6** Multiple inheritance – UML diagram

### **Example 3.5: Example for Multiple Inheritance in UML/C++**

We know multiple inheritance means deriving from more than one base class. For example, refer to Figure 3.7.



**Figure 3.7** Java's solution for ambiguity problems of C++ multiple inheritance

## Implementation in C++

---

```
class Student { ... };
class Athlete { ... };
class Professional : public Student,
public Athlete { ... };
```

---

## Implementation in Java: NOT supported

### 3.8 Interface

As shown above, Java language has solved the problem ambiguity in multiple inheritance by simply not allowing multiple inheritance feature at all. Instead, it solves this problem by allowing only one inheritance, i.e. single base class and multiple interfaces. An interface provides

names of the functions it provides service and a class can implement an interface which in turn has functions implemented in one of the derived classes of the interface.

### **Example 3.6: Example for Interface in Java**

Objects need to interact with users and the outside world. Java's interface is used by object to interact with the outside world. An interface simply will have names of all the functions for which the object provides functionality. Only names but no implementation. Therefore, we can say that interface is function names with empty bodies. For example, a Student objects interface is shown below:

---

```
interface Student
{ void GetData();void
ComputeResult();void Displayresult();}
```

---



Suppose name of our class is `UGStudent`, then we can use the interface `Student`, as shown below:

---

```
class UGStudent implements Student
{ // class definition here }
```

---

We can say that interface is a contract between the class and outside world regarding providing certain functionality.

### 3.9 Access Specifiers in Inheritance

Inheritance allows one data type to acquire the properties of another data type. But this facility largely depends on access specifiers used while inheriting. The access specifiers are public, private and protected.

- **Public Inheritance:** Out of the three specifiers, public is most widely used. Indeed, when we say inheritance, we mean public inheritance. In public inheritance, all non-private members, i.e. public and protected members, are available to descendant classes.
- **Private Inheritance:** This is less frequently used. Here only functionality is available to descendant classes but not attributes. For example, it is possible to fly like a bird by acquiring function `FLY()` through private

inheritance, but one cannot become a bird by acquiring the attributes like colour, beak, wings, etc.

- **Protected Inheritance:** This feature is least used since protected means public to descendant classes and private to others.

When access specifier is omitted, a class inherits privately, since only public accessory functions defined in the class become available to the descendant and not attributes, thus satisfying the definition of private inheritance. User-defined data type **struct** has no security access specifiers. Therefore, struct inherits publicly by default.

### 3.10 Run-time-type Information – RTTI

**Dynamic\_cast operator**, which allows the program to safely attempt conversion of an object into an object of a more specific object type. This feature relies on run-time-type information (RTTI).

**Static\_cast operator:** Objects known to be of a certain specific type can also be cast to that type with `static_cast`, a purely compile-time construct.

When base class function is overridden, exactly which of the base class function to call depends on the type of object. It is NOT possible to determine the type of the object at compile time. Hence, a pointer to base class, depending on the type of object that will be known only at run-time, dispatches the correct method. This feature is called **dynamic method dispatch in Java or virtual function methods in C++**.

### 3.11 Virtual Functions

In inheritance relation, the derived class gets access to protected member data through public accessory functions thereby achieving an important objective-oriented programming facility called code reusability.

But what about object to base class using the member functions belonging to derived class? This feature facilitates object to base class to execute any one of the derived class functions, depending on the user's choice at run-time.

This means that object to base class is provided with a bridge to selected derived

class function and hence it can execute the function. The bridge is made available when we declare a virtual function in the base class and derived class overrides base class function.

If function is declared as virtual function in base class, we can execute an overriding function with the same name in the derived class with a pointer to base class. Pointer to base class is provided by virtual function. In summary, we can say that from base class we can execute any function with the same name as that of the virtual function in the base class, depending on the user's choice at run-time.

The above feature has been facilitated by C++ by providing variable pointers to a base class. These variable pointers can refer to objects of any derived classes of that type in addition to objects exactly matching the variable type. This allows arrays and other kinds of containers to hold pointers to objects of differing types. Because assignment of values to variables usually occurs at run-time, this is necessarily a run-time phenomenon. This feature facilitates

run-time polymorphism and dynamic data binding explained in the next section. As the type of an object at its creation is known at compile time, constructors, and by extension copy constructors, cannot be virtual. If any functions in the class are virtual, the destructor should also be defined as virtual.

### 3.12 Pure Virtual Functions

C++ supports another variation of virtual functions like pure virtual functions. Virtual function can be converted by writing `= 0` at the end of the function declaration:

---

```
virtual void FindArea() = 0;
```

---

Note that objects cannot be created for a class with pure virtual functions and such classes are called abstract data type (ADT). A member function can also be made “pure virtual” by appending it with `= 0` after the closing parenthesis and before the semicolon. Objects cannot be created of a

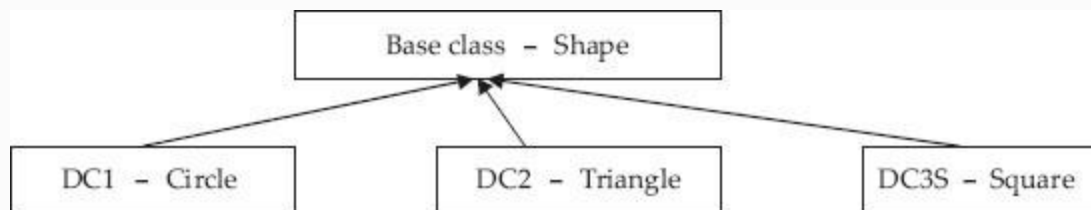
class with a pure virtual function and are called ADTs. Any derived class inherits the virtual function as pure and must provide a non-pure definition of it (and all other pure virtual functions) before objects of the derived class can be created. An attempt to create an object from a class with a pure virtual function or inherited pure virtual function will be flagged as a compile-time error.

### 3.13 Run-time Polymorphism and Dynamic Data Binding

Inheritance solves the problem of reusability. That is, derived class can access all the data members and function members of base class that are declared as protected. We have also learnt that derived class can override the functions defined in base class. Further, we have seen while dealing with virtual functions that with a pointer to base class we can call the derived class object. Combining the above two features of OOP language gives us a powerful tool, called

run-time polymorphism and dynamic binding.

Refer to Figure 3.8 which shows hierarchical inheritance with virtual functions defined in base class and each of the derived class overriding the base class virtual function with its own implementation.



**Figure 3.8** Hierarchical inheritance

What this feature means to the programmer is that with a pointer to base class, we can decide at run-time to call particular derived classes overriding function. In other words, we can bind the overriding function from several of the derived classes with a pointer to base class. In the previous section we have seen the

working of virtual functions with which we can achieve binding of derived class functions with a pointer to base class.

### 3.14 Class as Abstract Data Type

A class in object-oriented languages can also be called **ADT**.

The features of OOP language such as run-time polymorphism and dynamic binding will allow us to define a base class with virtual functions with no implementation (called pure virtual functions) or with dummy functionality just to indicate to the user that implementation is by derived class. These classes are called ADTs because they hide the implementation.

Refer again to Figure 3.8. We will declare a base class (ADT) called Shape. You will appreciate Shape has no definite shape, no definite area or perimeter or specific draw routine. Shape will hence declare virtual functions with only names and no implementation details. We will derive classes like Circle and Square and provide the **solid** implementation of these virtual



functions. Solid means, all virtual functions will be implemented in each of the derived classes.

### 3.15 Separation of Behaviour and Implementation

ADT or base class with virtual function with no implementations is also called interfaces in some OOP languages. The base class will simply work as interface, informing the user about the functionality available, through definition of virtual functions and depending on the user's choice, the particular implementation from any one of the derived class is executed. The user is hidden from the details of implementation. The code of actual implementing function is linked to calling function only at run-time.

### 3.16 Decoupling

In inheritance, we have reusable code. Encapsulation means binding data and code. When run-time polymorphism is being used, it is best to polymorphically decouple the

encapsulation to prevent discrete code modules from interacting with each other. In practice, you will notice that base class overridden function is automatically hidden and can be called explicitly by referring to the base class function.

### 3.17 Standard Template Library

One of the true definitions of objective-oriented programming is the ability of the programmer, through language, to use library facility. C++ has much developed Standard library in the form of stream IO library that provides extensive functionality for input and output. Standard template library (STL) is the most significant addition to library provided by C++. Using the module in STL, a programmer can write code using data structures and algorithm implementation for data structures like vectors, lists, queues, maps, etc.

### 3.18 Summary

1. Extensibility and reusability of OOP languages have been realized with language features like inheritance, virtual inheritance, standard templates, etc.
2. Containment represents a **has** type of relation. Supported by C++ but not supported by Java. A class contains an instance of another class.
3. Containment is also called composition and aggregation in the literature.
4. Base class is called superclass in Java. Derived class is termed as subclass. In inheritance, all non-private members are available to descendants. That is, base class is generalization and derived class or subclass is specialization.
5. In inheritance, subclass functions can override base class overloaded functions. If subclass function overrides only one of the overloaded functions of base class, then balance overloaded functions will be truncated and will not be available to base class.
6. Inheritance types supported by C++ are
  - Single- and Multilevel inheritances
  - Hierarchical inheritance
  - Multiple and hybrid inheritance (not supported by Java)
7. Multiple inheritance is deriving a class from more than one base class.
8. Base class and derived class are called superclass and subclass in Java.
9. Subclass methods can override base class functions. If such overriding takes place of overloaded base class functions, then balance of overloaded functions are truncated and they are not available to subclasses.
10. If more than one subclass is derived from the same base class, then the base class is called a virtual base class.
11. If a subclass derives from more than one base class, then to resolve the ambiguity arising out of the multiple inheritance relation, both the base classes must be

- declared as virtual. Inheritance is called virtual inheritance.
12. Java solves the problems of ambiguity due to multiple inheritance type of relations by allowing only single inheritance but several interfaces. The interfaces contain only names of the functions and hide implementation of these functions. **Implementation of an interface is also called realization.**
  13. **Dynamic\_cast operator**, which allows the program to safely attempt conversion of an object into an object of a more specific object type. This feature relies on RTTI.
  14. **Run-time polymorphism** means that with a pointer to base class, we can decide at run-time to call particular derived classes overriding function. In other words, we can bind the overriding function from several of the derived classes with a pointer to base class.
  15. **Virtual Function:** If function is declared as virtual function in base class, then we can execute an overriding function with the same name in the derived class with a pointer to base class. Pointer to base class is provided by virtual function.
  16. **Pure Virtual Function in C++:** Pure virtual functions are virtual functions with "=0" attached at the end. No object can be created for a class with pure virtual functions defined in it.
  17. Java supports interfaces to implement the concept of pure virtual functions of C++.

## Exercise Questions

## Objective Questions

1. A subclass can have access to private data members of superclass – TRUE / FALSE
2. Generalization means inheritance – TRUE / FALSE
3. Specialization refers to
  1. Base class
  2. Subclass
  3. Superclass
  4. None
4. Which of the following refer to containment:
  1. Composition
  2. Aggregation
  3. Has a
  4. Is a
  1. i and ii
  2. i, ii and iii
  3. i, ii and iii
  4. i, ii and iv
5. Multiple and hybrid inheritance
  1. Supported by C++
  2. Supported by C++ and Java
  3. Supported by Java
  4. Supported by none
6. If a subclass is derived from more than one base class, then the inheritance type is
  1. Multiple inheritance
  2. Virtual inheritance
  3. Virtual base class
  4. Multilevel inheritance
7. Realization can be achieved by
  1. Extending a base class
  2. Implementing an interface
  3. Virtual functions
  4. None of these

8. When a subclass redefines a base class function using the same name and signature, it is called

1. Base class overloading
2. Base class function override
3. Base class function hiding
4. None of these

9. Composition is implemented using

1. Has a
2. Is a
3. Interface
4. Virtual function

#### **Short-answer Questions**

10. Explain composition / containment features of OOP languages.
11. Generalization / specialization means inheritance. Do you agree?
12. Distinguish between composition and inheritance.
13. Why does Java not support containment?
14. What are the types of inheritance supported by Java?
15. What is the alternate provided by Java to multiple inheritance of C++?
16. What is the difference between a class and an interface in Java?
17. Can constructors be declared as virtual? Why or why not?
18. Can destructors be declared as virtual? Why or why not?
19. Distinguish overloading and overriding in inheritance relationship.
20. What is an interface in Java?
21. Explain access specifiers and their relevance in inheritance relation.
22. Explain RTTI and dynamic cast operator.
23. What are virtual functions? How are they useful?
24. What are pure virtual functions? Why can objects not be instantiated to class with pure virtual functions defined

inside?

#### **Long-answer Questions**

25. What are the various types of inheritance supported by OOP language?
26. What are run-time polymorphism and run-time data binding?
27. ADT / interface are supporting the concept of separation of behaviour and implementation. Do you agree?
28. What is a virtual function? Why do we need virtual functions? What are the rules for virtual function implementation?
29. What is an abstract class? Why do we need abstract classes? What are the rules governing virtual base class?

#### **Assignment Questions**

30. Write inheritance hierarchy for the animal kingdom.
31. Explain polymorphism with a suitable example.
32. Explain generalization and specialization with suitable examples.
33. Explain the terms aggregation and separation with suitable examples.
34. Explain various types of inheritances.
35. When do you use inheritance and on what occasions do you use interface?

### **Solutions to Objective Questions**

1. False
2. True
3. b
4. c
5. a
6. a
7. b

8. b

9. a



# 4

## Dynamic Modelling

### LEARNING OBJECTIVES

*At the end of this chapter, you should be familiar with concepts and usage of*

- Static characteristics of a system. Use of UML static diagrams such as class, object and package diagrams.
- Events and States in a system and their role in depicting dynamic behaviour.
- Use UML diagram to design the system using OOPS concepts.

### 4.1 Introduction to Static and Dynamic Modelling

While modelling an object, we are interested in two aspects. The first is the structure.

When we say structure, we mean it is defined at compile time. We can also call this a static characteristic. For example, when we define a class, it includes attributes, behaviour in terms of attributes, links and associations, and some complex structures such as class within a class definition. We use class diagrams, object diagrams and package diagrams to describe the static characteristics of a system model.

Objects are instances of classes. Classes have attributes and behaviour. We are aware that instances are created at run-time, and these objects have a specific lifetime. They also undergo changes based on messages received during an object's lifetime.

Methods or functions, as they are otherwise called, are responsible for an object's behaviour. Attributes hold values that can be changed during the lifetime of an object.

**State of an object** is defined as **current values** held by the attributes of an object. It is the result of behavioural changes of an object till that particular instance.

In this chapter, we would concentrate on static and dynamic modelling of object. Firstly we provide over view of UML, a language used for describing the behaviour of objects. We have used class diagrams, object diagrams, package diagrams to describe the static characteristics of a system modelling. We will also introduce dynamic modelling through study of collaboration diagrams, sequence diagrams, activity diagrams, and state transition diagrams etc.

## 4.2 Lifetime of an Object

Objects are dynamic entities. They are created at run-time as per the need of the program when the program is under execution. We say an object is instantiated. Constructor does this job of instantiation and also allocates initial values to the attributes of an instance of a class. Objects have attributes and behaviour. An object can communicate with other objects by passing messages. An object passes information about its state to others through **get** and **set** functions that are defined in a class. Get

functions that return a value of attributes are also called assessors. Set functions that allow attributes to be modified are also called mutators.

Objects in Java use references to uniquely identify the instance of an object, whereas C++ uses pointers and references.

Once the object completes the role assigned to it, it has to be deleted from the memory so that the freed memory can be dynamically allocated to other objects. This is done through garbage collector in Java. Garbage collector is a program initiated by the Java run-time system to remove objects that are no longer required. In C++, programmers can explicitly remove an object through the use of delete function.

## 4.3 Unified Modelling Language – A Tool for OOAD

### *4.3.1 Overview of UML*

The Unified Modelling Language (UML), is used as a graphical tool in object-oriented analysis and design (OOAD) methodology to

model a software-based system under development.

When do we resort to modelling?

- When we would like to model the system and study its behaviour.
- When we further wish to simulate the system and study its performance and operations.
- When the system under development is complex and not a trivia.
- When the cost of development of a software system is prohibitive.

What do we look in a modelling system?

- It should closely follow the development model we choose, for example, the Waterfall model.
- It should be graphical in nature.
- It should model all facets of Software Development Life Cycle.
- It should accommodate all popular and well-accepted theories of object-oriented technologies such as those of Grady Booch, James Rumbaugh and Ivar Jacobson.

The UML, founded by a consortium of industries such as Hewlett-Packard (HP), International Business Machines (IBM), Oracle, etc., under the aegis of a controlling group called Object Management Group (OMG), satisfies all the aforementioned criteria and is indeed an industry standard for software development. The UML can be

effectively used to conceptualise, modify, construct and deliver an object-oriented software system, including full-fledged documentation.

The term “Unified” in UML implies that UML unifies the theories of three leading scientists at a software company called Rational Software, Grady Booch (Booch methodology), James Rumbaugh (object-modelling technique [OMT]) and Ivar Jacobson (use cases and objectory), into a rational unified process (RUP). Later, RUP was merged with development methodologies such as Booch methodology, OMT and object-oriented software engineering (OOSE).

#### 4.3.1.1 Modelling

The UML is a graphical representation of a system under development. In addition, it also contains documentation such as use cases that are used for developing the system. The UML has two different views as described below:

- **Static view:** It provides the structure of the system under development in terms of objects, attributes and

operations. These aspects are depicted by class diagrams and object diagrams.


- **Dynamic view:** It depicts the behaviour of the system. It includes collaboration diagrams, sequence diagrams, activity diagrams and state transition diagrams.

The UML Version 2.2 has 14 diagrams, seven static diagrams and seven behavioural diagrams. Behavioural diagrams include four interaction diagrams.

#### 4.3.1.1.1 Structure Diagrams

Structure means the essential things that must be present so that the system can function as per specification. For example, the backbone and jaw are essential in human body to maintain the erect posture and to chew the food. Structure diagrams, as they depict the elements that are present in the system, represent the architecture of the system as shown below:

- **Class diagram:** It describes classes, their attributes and relationships among the classes.
- **Component diagram:** It shows the decomposition of software into components and the interdependencies between these components.
- **Composite structure diagram:** It describes the internal structure of a class and the collaborations that this structure enables.

- **Deployment diagram:** It describes the hardware used in system implementations, execution environments and artefacts deployed on the hardware.
- **Object diagram:** It is an instance of a class diagram at a particular instant of time.
- **Package diagram:** It describes how a system is split into logical groupings by showing the dependencies among these groupings.
- **Profile diagram:** It shows stereotypes as classes with <<stereotype>> and packages as <<profile>> stereotype. The symbol is  and indicates that the meta model is extended by a stereotype.

#### 4.3.1.1.2 Behaviour Diagrams

Behavioural diagrams depict the functionality of the system. The different types of behaviour diagrams are shown:

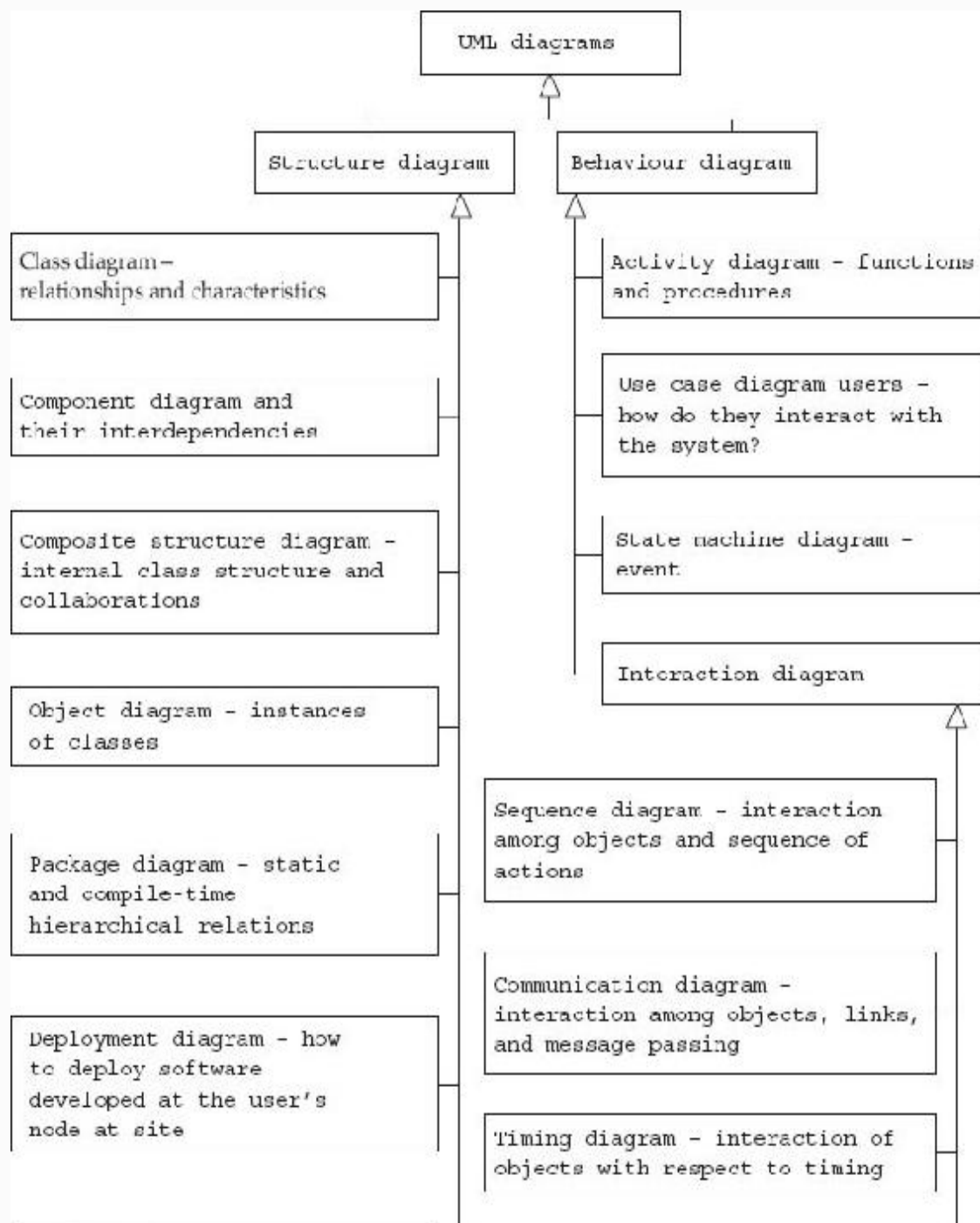
- **Activity diagram:** It shows the overall flow of control.
- **UML state machine diagram:** It depicts the states and state transitions of the system.
- **Use case diagram:** It describes the functionality provided by a system in terms of actors.

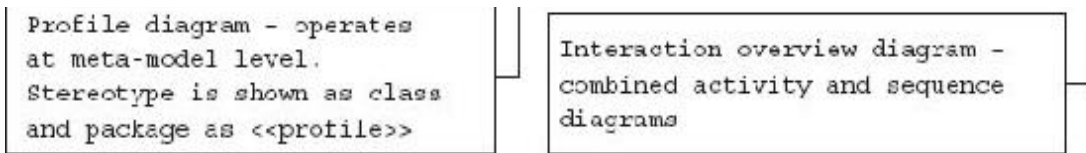
#### 4.3.2 Classification of UML Diagrams

The classification adopted by UML is shown in [Figure 4.1](#). Class, object and package diagrams depict the structure and static characteristics. Use case diagrams, interaction diagrams, state diagrams and activity diagrams, on the other hand, depict



behavioural modelling. In Chapter 2, we have dealt in detail with structure. In the following sections, we discuss dynamic behaviour.





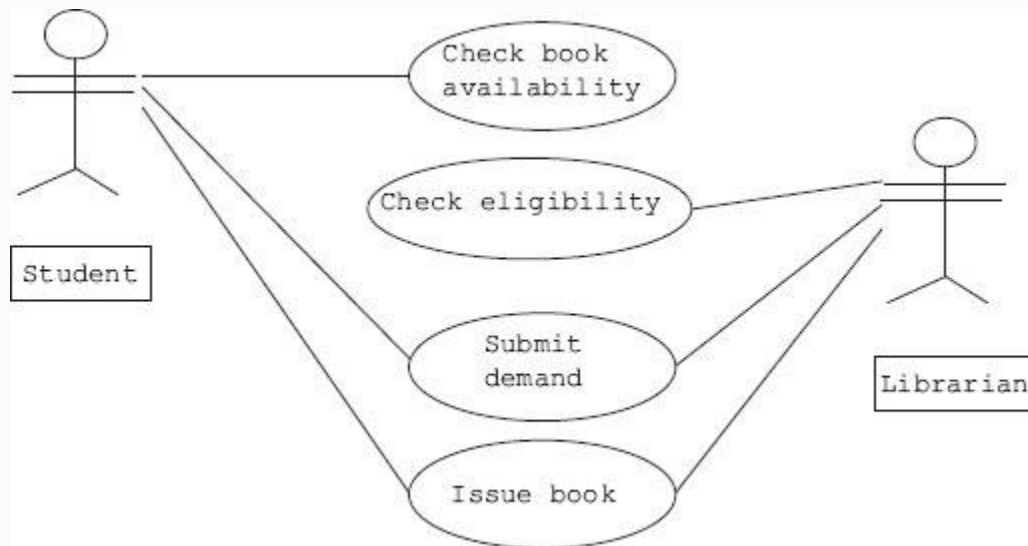
**Figure 4.1** Classification of UML diagrams

## 4.4 User Requirements – Use Cases

We want to produce a system that the user wants. Once the system is produced, users interact with it to produce the desired result. So we need some description on how to go about it. **Use cases** typically provide such description. Use cases describe what the system does from the point of view of an external user. They provide a **sequence of steps** describing the interaction between the system and the user. This sequence of steps is called a **scenario**.

For example, for an automated library information system in a college, a scenario could be as follows (Fig. 4.2): A student browses online the catalogue of books available in the library and selects the accession number of the book he or she wishes to borrow from the library. The issue subsection of the automated system checks the entitlement of the student and confirms the issue of the book to the

student. The system issues an authorization slip to the issue gate by email. The student carries the book to the issue gate, where the in-charge affixes the authorization slip on the inside-cover page.



**Figure 4.2** Use cases in an issue section of a library system

A second scenario can be that authorization fails because the student has exceeded the limit of borrowing.

A third scenario could be that a student has failed the identification checks.

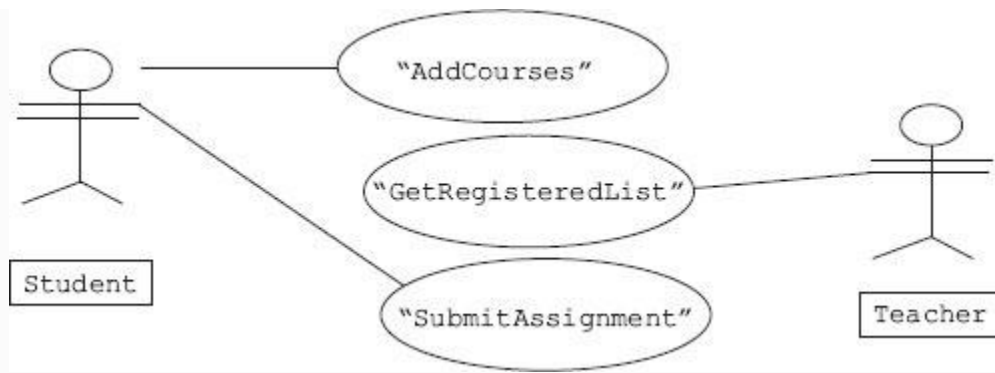
There are three scenarios described for borrowing a book from the library; but the goal of the student in all three scenarios is

the same, i.e. borrowing a book. So, we can now say that a use case is a collection of different scenarios but with a common goal.

In a use case, we call users as actors. An actor can be a human being, a device, or a software module. For example, banking software system manager, customer, smoke sensor and software agent working in the background are all actors. A use case can have several actors and conversely one actor can use several use cases. An actor is shown as a stick and a use case as an oval. The UML has a use case diagram that shows how use cases are related to each other.

As a second example consider the case of a student enrolling for a third semester course.

The actors and use cases involved in this example are shown in Figure 4.3.



**Figure 4.3** Use cases in a student registration process

Use cases explain functional requirements of a system under study. Each use case is required to be described in detail before implementing. A scenario explaining the step-by-step procedure of how the user gets what he or she wants out of a system is given below:

**Example 4.1: Scenario from the use case of a student registration process**

Gautam enrolls for third semester course. A step-by-step procedure is shown below:

1. Gautam goes to the registration window and enters his identification details.
2. Computer gives a list of things Gautam can do.
3. Gautam selects registration for third semester courses.
4. The screen shows the list of courses and open electives.
5. Gautam selects courses and open electives.
6. The system confirms Gautam's registration for third semester.
7. Gautam logs out.

Each use case will have a primary actor that calls on the use case to deliver the functionality expected. A use case can have another use case defined inside. Each step in a use case is an interaction between the actor and the system. In addition, each use case can have **preconditions, a guaranteed result, and also a trigger** that sets the use case into action.

#### 4.5 Architecture and Domain: What Are They?

In any real-world problem, we have users and their wants and needs. Accordingly, we

define objects and their interactions, i.e. we decide architecture. Users' world is called **Domain**.

#### *4.5.1 Architecture*

We have users of the system and use cases. We have classes in the software we are developing. Classes are a reflection of the real-world client (users) and what they want (use cases). Based on clients and their real-world interactions, we define interaction between objects and classes as links and associations to deliver functionality. This arrangement is termed as **architecture**.

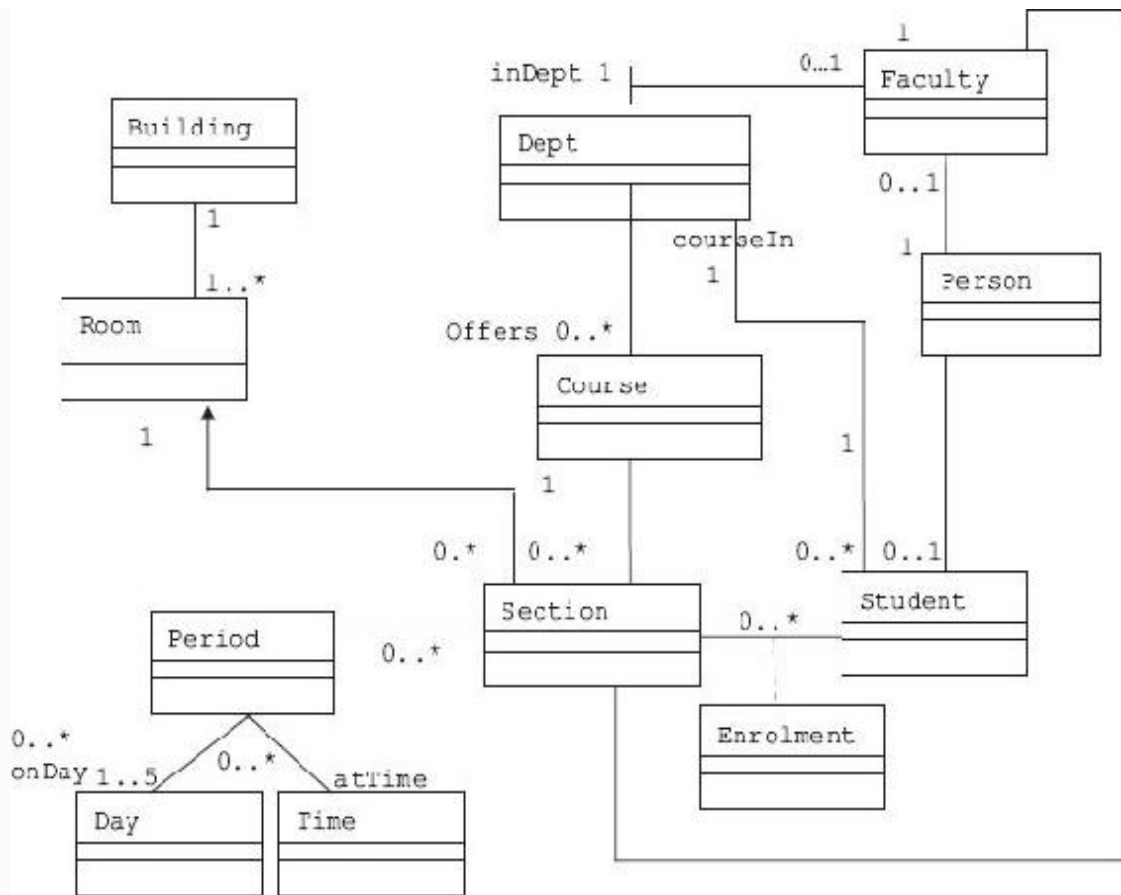
We can define architecture as a depiction of objects and their attributes together with the operations these objects perform to meet user specifications or needs.

#### *4.5.2 Domain*

The clients and their interactions exist in the real world. Class is a conceptual model that maps real-world problems to software. The users' world is called **domain**.

We will show the users domain and our architecture as a collection of square boxes that contain names of the objects (Fig. 4.4). We will also show their links and associations by drawing interconnecting lines. As we have studied in Chapter 2, we can also amplify by specifying role names and multiplicities. In Example 4.2, we consider domain and architecture of a college.





**Figure 4.4** Domain and architecture diagram for a college

The square boxes you see are called entities. For a complex project, attributes of each entity is maintained in a table. This table is called a **data dictionary**. Before the advent of the objectoriented programming paradigm, these diagrams

were also called **entity-relation diagrams**.

### **Example 4.2: Domain and architecture of a college**

In order to get an insight into which object does what, it is advisable to list out the sequence of steps for the entire process. Let us revisit the overall sequence of operations for the registration process we have discussed in Example 4.1, with amplification shown to registration process as shown below:

1. Gautam goes to the registration window and enters his identification details.
2. Computer gives a list of things Gautam can do.
3. Gautam selects registration for third semester courses.
  - 3.1 Add a course.
  - 3.2 Delete the course already added (update).
4. The screen shows the list of courses and open electives.
5. Gautam selects courses and open electives. He has the option of adding a course or dropping a course as update feature.

6. The system confirms Gautam's registration for third semester.
7. Gautam logs out.

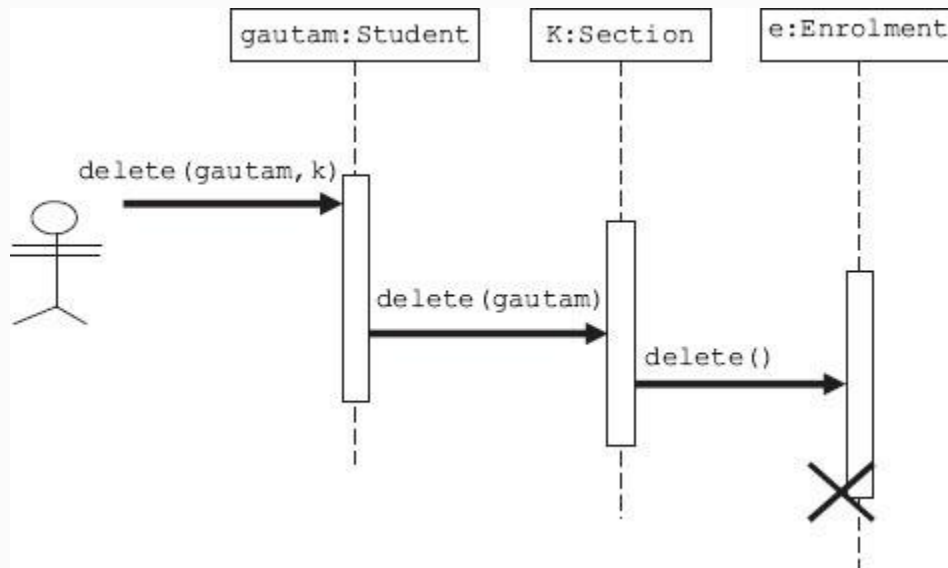
Then for each step in this sequence of steps, we can prepare a sequence diagram or a communication diagram and/or timing diagram. What each object does is an important issue for the user. UML provides sequence diagram and communication diagrams to solve this issue. Timing diagram is a variation of the sequence diagram that specifies against time base.

## 4.6 Sequence Diagram

Users are represented by stick figure. Each object is represented by a square box (object representation). A vertical line is drawn from each object to represent the activity base line. Arrows emanating from each vertical line show the object sending a message or calling a function belonging to some other object.

In Figure 4.5, we show the sequence diagram. The actor understudy is a third-

year student. The course coordinator would like to drop or delete Gautam's name from section K. The sequence of activities is as follows:

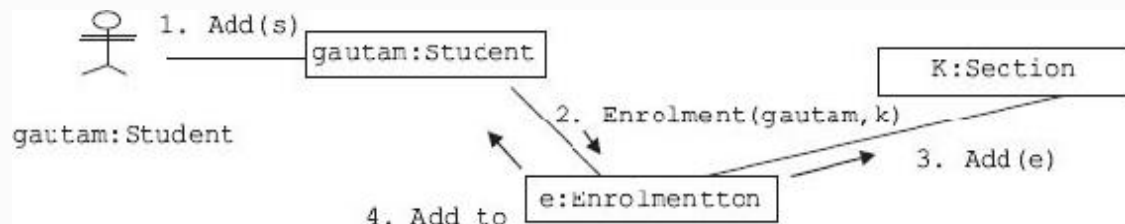


**Figure 4.5** Sequence diagram for deleting a name from registration process

## 4.7 Collaboration (Communication) Diagram

Earlier, communication diagrams were known as collaboration diagrams. These are essentially the same as sequence diagrams except in the case of communication

diagrams. The time of occurrence of messenger, called time ordering, is shown explicitly by numbering the messages as shown in Figure 4.6. Each message in a collaboration diagram has a **sequence number**. The top-level message is numbered 1. Messages at the same level (sent during the same call) have the same decimal prefix but different suffixes of 1, 2, etc., according to when they occur.



**Figure 4.6** Communication diagram for registration of a student

The key elements of a communication diagram are

- **Objects as classifier:** They are shown as rectangular boxes. The object-role rectangles are labelled with either class or object names (or both). Class names and object names are separated by colons (:).

- **Connector:** This is a line indicating a message.
- **Named arrow:** This represents the sender, receiver.

In order to read the collaboration diagram, simply start at the message numbered as 1 and follow numbers. In Figure 4.6, we have used a simple numbering scheme. However, UML uses a nested numbering system such as

Simple numbering scheme: 1, 2, 3, 4, ...

Nested UML numbering: 1, 1.1, 1.1.1, 1.1.2,

...

Collaboration diagrams convey the same information as sequence diagrams; but they focus on object roles instead of the times that messages are sent.

## 4.8 Events and State

Class diagrams are static in nature and depict the links and associations between classes. They cannot be used to depict the behaviour of objects during objects' lifetime.

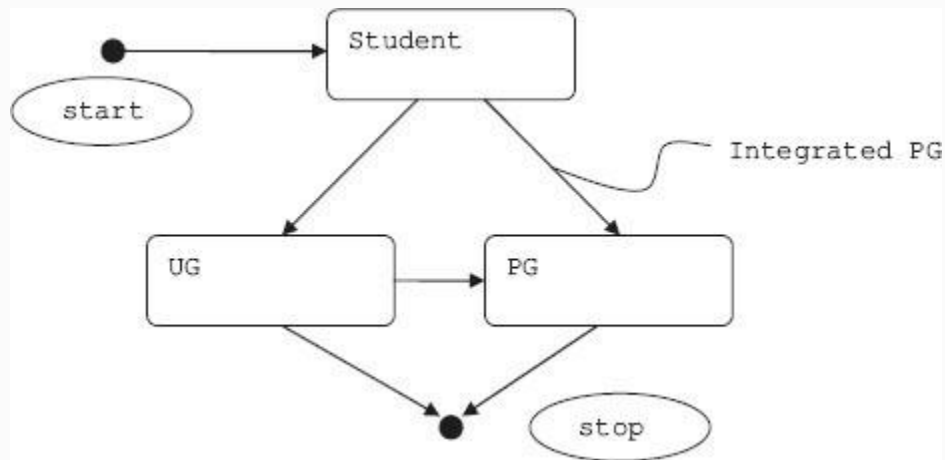
Objects have behaviour and state. The state of an object is defined by current

values of attributes. We have further learnt that access specifiers like getters and setters can change the values of attributes. We can say that some events trigger the process of changing the values of attributes and thereby change the state of an object.

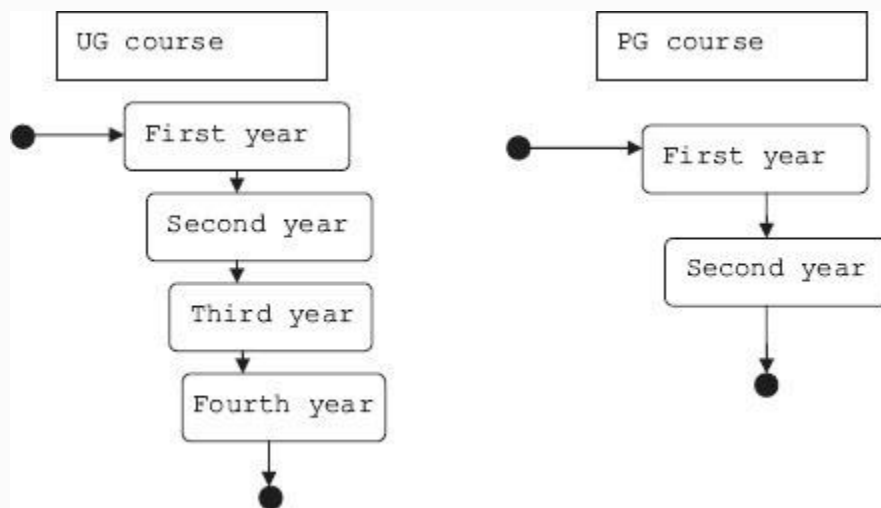
Note that a state diagram describes behaviour of class objects. Therefore, all objects share a single state diagram. But the path object follows in the state diagram is specific to an instance of the object.

The state chart notation has been developed in computer science as part of theory of automatic machines called the automata theory. This science has been adopted by UML to depict the way in which an object changes. Each change in behaviour is modelled as a change in state.

A state diagram represents events and states. When an event occurs, transition to the next state depends on current state and event. The state diagram shows these changes in states and allows changes that cause the change of state. As an example let us see how a student enrolls in a college. The state diagram is shown in Figure 4.7.



**Figure 4.7a** State diagram for student pursuing a professional course




**Figure 4.7b** Student undergoing UG/PG courses



In a state diagram, observe that start and stop are shown as black blobs. Each state is shown as a rectangle with rounded corners. Change in state means that the object has changed its behaviour. For example, a student when he or she undergoes the undergraduate (UG) course, i.e., first- to fourth-year courses, and qualifies, then he or she becomes a UG. A student can enrol either for UG course or directly for an integrated postgraduate (PG) course. Each arrow represents a change. We can add a lot of additional information on the state chart diagram:

- It can be labelled with the name of the function that causes the change.
- Add operations or events or conditions that trigger changes in the object's state.
- Add operations to be executed while entering and leaving the state.

### **Example 4.3: State chart diagram for online Internet banking**



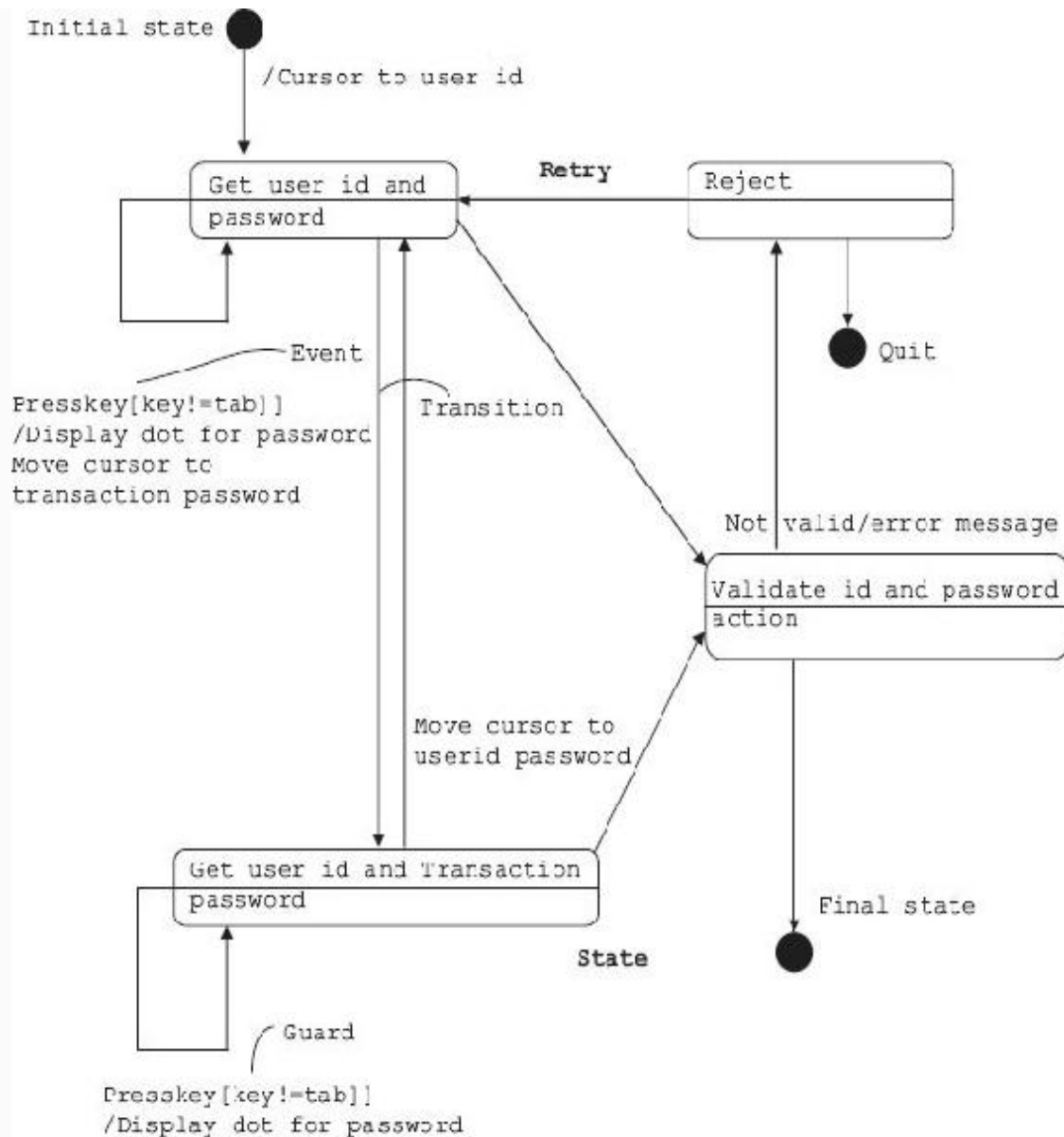
We are now ready to define a dynamic model as a collection of all state diagrams that interact with each other with interacting events.

In online banking, there are the following distinct states:

- Logging into the system by entering username and password
- Validating username and password
- Getting username and transaction password
- Validating username and transaction password

From each state, transactions emerge.

Transactions are shown as arrows going from one state to another state. Events that trigger transition from one state to another are shown next to the arrows. In Figure 4.8, in our system we have two self-transitions that occur when the user enters invalid username or password. Till user does not enter tab key or enter key the cursor remains and dot is displayed to maintain secrecy of data being entered.



**Figure 4.8** State diagram for Internet banking

The Internet banking system obtains user id and Internet password and validates

them. If they are found to be correct, user can enter transaction details. The system, as a double safety measure, asks for user id and transaction password. These entries are also subjected to validation. If they are found to be correct, final state is reached wherein user can attend to his or her transaction or it is rejected.

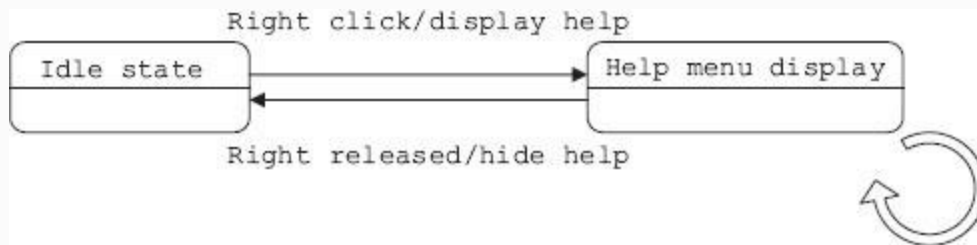
The start and stop are shown as black circles and are dummy states. Each state can have an action. The object performs the action, and as a result of this action the object can change its state.

#### *4.8.1 Activity/Action and Operations*

States perform activities. It can be a single activity or a sequence of activities. The notation used is "Do X1". If a state has "Do X1" mentioned in rectangular box, it means that on entry it will do activity X1.

On the other hand, an action signifies immediate operation. Operations are indicated with a "/". For example, we have shown / Display Dot as operation in the Internet banking example in Figure 4.8.

Action can also be used to generate other events. For example, from an idle state, by right clicking the mouse button, we can generate a help menu. On release, the help menu can be removed. The state diagram is shown in Figure 4.9.



**Figure 4.9** Example of internal actions

### *4.8.2 Nested State Diagrams*

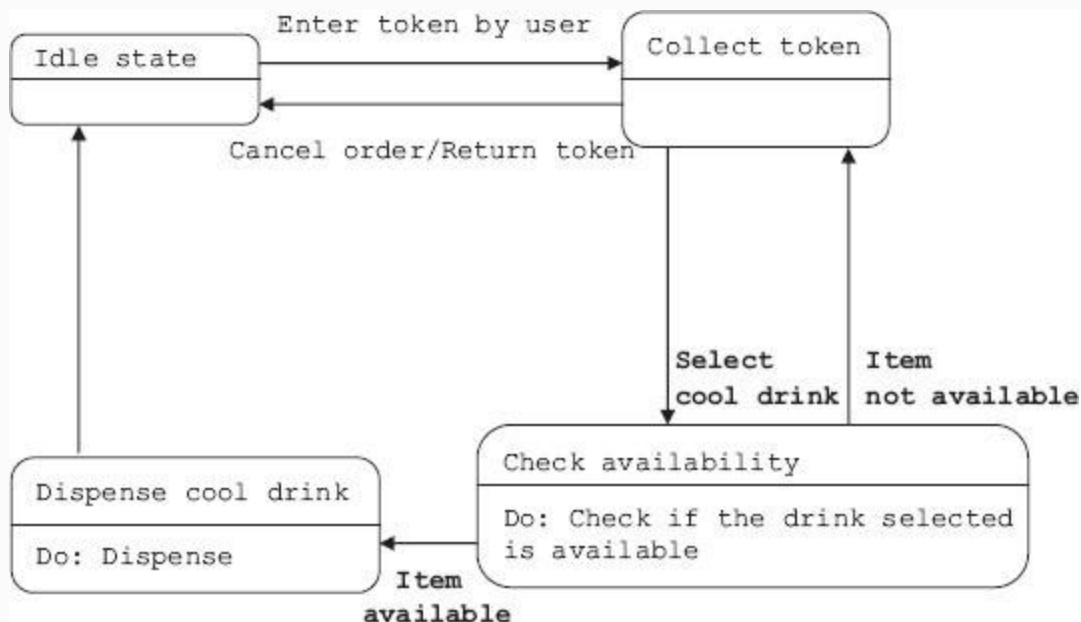
We have learnt that states perform activities. Activities can be broken down into subactivities. Now, if each subactivity represents a state, we can say state is a collection of substates and we can call the structure as a nested state diagram.

As an example, consider the cool-drink vending machine installed at your campus.

The main activities involved in obtaining a cool drink from the machine are as follows:

- User enters token.
- Validate token.
- User selects brand of cool drink.
- Dispense the drink.

The vending machine model is presented in Figure 4.10. The main states shown are as follows:

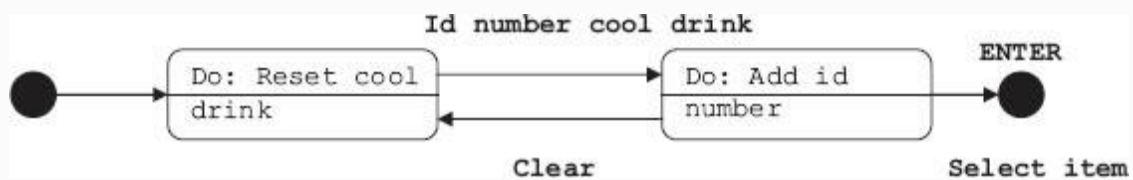


**Figure 4.10** Soft-drink vending machine model

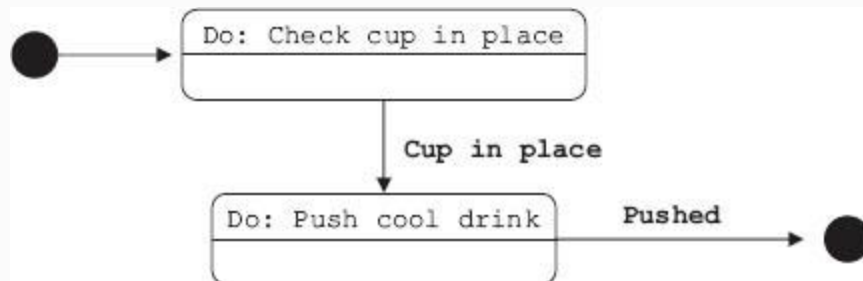
- Collect token from the user.
- Check availability.

- Dispense item.

The state diagram contains an activity called dispense item and an event called select cool drink. We will expand and show these two through nested state diagrams. Figures 4.11a and 4.11b show nested state diagrams.



**Figure 4.11a** Expanded state diagram for the event select item



**Figure 4.11b** Expanded state diagram for dispense item

For selecting an item from an available list of cool drinks, user has to enter the identification number for the cool drink selected. He or she will have the option of cancelling the entry and re-entering the number. On pressing ENTER, the item gets selected and is registered for dispensing.

## 4.9 Activity Diagram

A state chart diagram tells us about an object undergoing a process. An activity diagram shows the flow of activities in a single process. It shows the way in which each activity is dependent on other activities. Therefore, we can define activity as a diagram showing the flow of activities and their interdependence on one another.

For explaining the concept of activity diagram, we choose the same example of interbank money transfer through the Internet as in Example 4.3.

The classes involved in the case of interbank money transfer through the Internet are **user, client computer system and bank**. The activities



concerning each class are shown as rounded rectangles in a separate column called **swim lane**. The salient features of an activity diagram are as follows:

- Swim lane identifies the object responsible for carrying out a particular activity.
- Each activity gives rise to a single transition, and each transition connects to the next activity.
- A transition, sometimes based on decision box, may branch into two or more transitions.
- Each transition coming out is indicated using a guard expression ( [ ] ).
- A branch and a subsequent merge are indicated using a shallow diamond.
- Sometimes, a transition may **fork** into parallel activities and later merge. These are shown as solid bars in Figure 4.11.

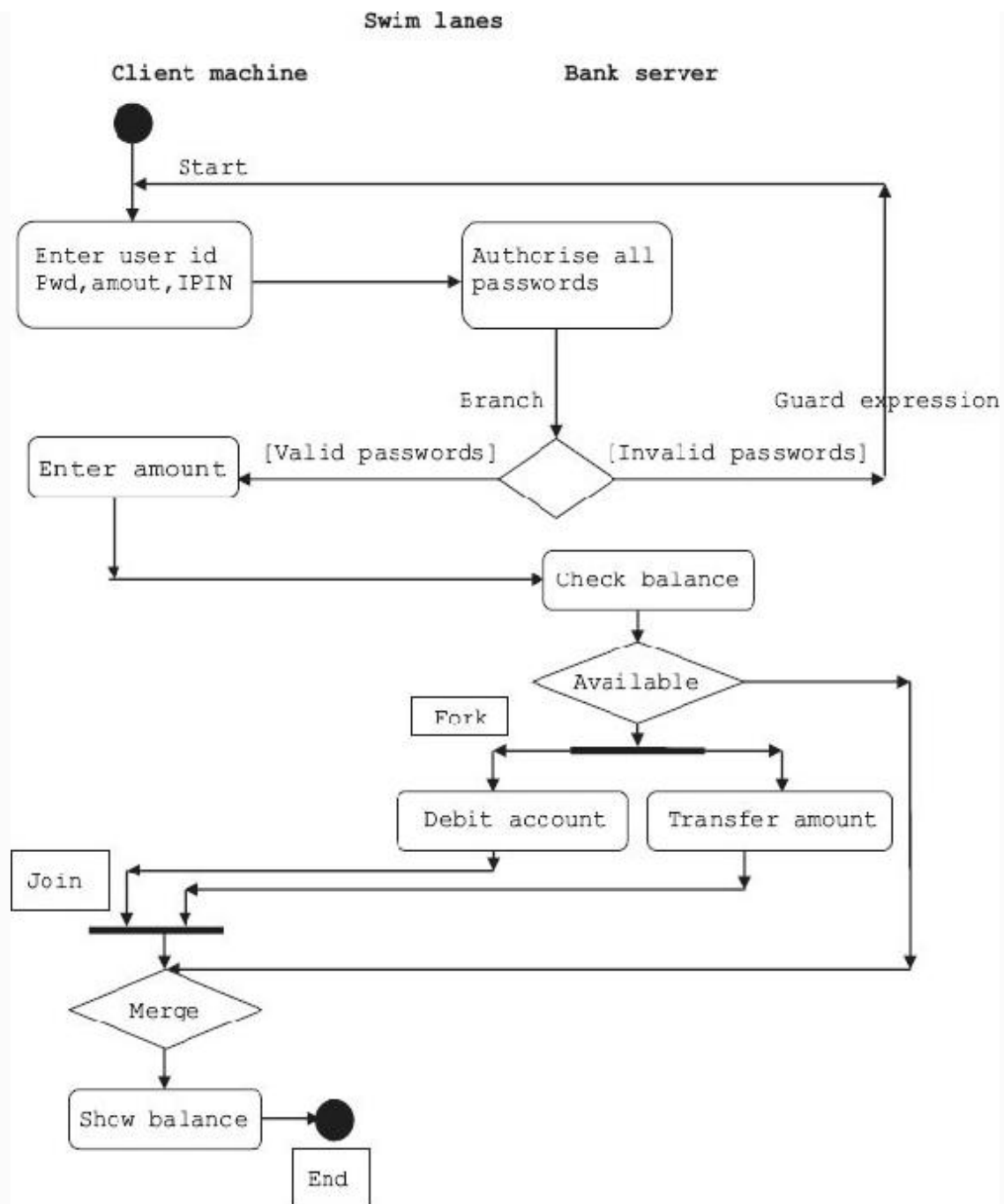
### **Example 4.4: Activity diagram for online Internet banking**

In money transfer using online banking, the activities involved are shown below:

- User logs into the system by entering username and password.

- Bank server (bank) validates username and password.
- User enters amount and transaction password (IPIN) and transfer details.
- Bank validates transaction password.
- On successful validation of IPIN, bank checks availability of funds.
- Bank transfers the amount to destination account.
- On confirmation from destination bank's server, bank debits the account.
- The transaction details and fund balance are displayed.
- User logs out.

The activity diagram in Figure 4.12 is self-explanatory and follows the list of activities described above. There are two lanes, one for bank server and another for client machine.



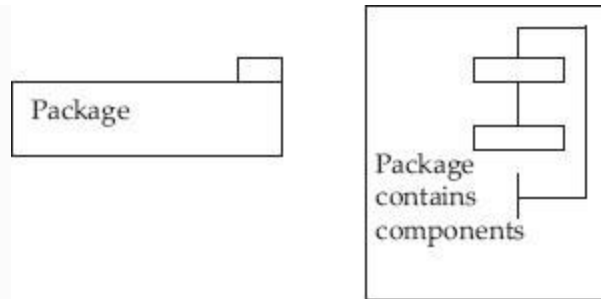
**Figure 4.12** Activity diagram for online Internet banking

◇ White diamond is used to indicate branching, and guard expressions are enclosed in [ ].

—— A thick line is used to indicate fork, i.e., to initiate more than one activity such as debit account as well as transfer amount. Same symbol is used for join activity also.

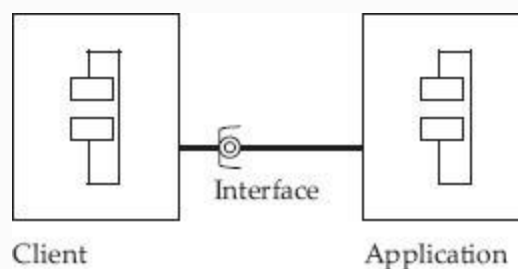
#### 4.10 Packages and Components – A Way to Organize Large and Complex Projects

You are familiar with files and folders. In order to organize your work, you open folders and within folders you store files. Package is like a folder. We can store all our classes in this folder called package directly. If the number of classes is very large, then classes can be grouped into components and components into packages (Fig. 4.13).



**Figure 4.13** Notation for packages

Components contain classes. They provide pluggable interfaces to users. We will use plug and socket concept to denote an interface. For example, in [Figure 4.14](#), Application module provides interface to students' module.

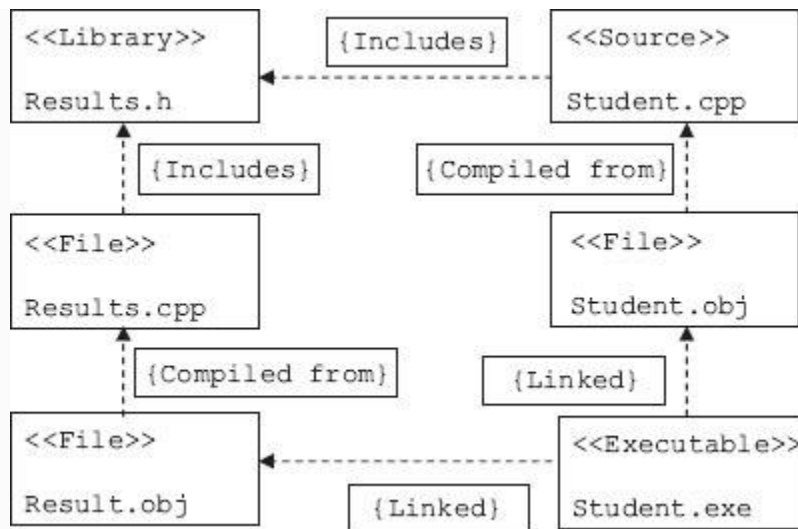


**Figure 4.14** Notation for components

If we code the classes in components, then these coded classes are called artefacts.

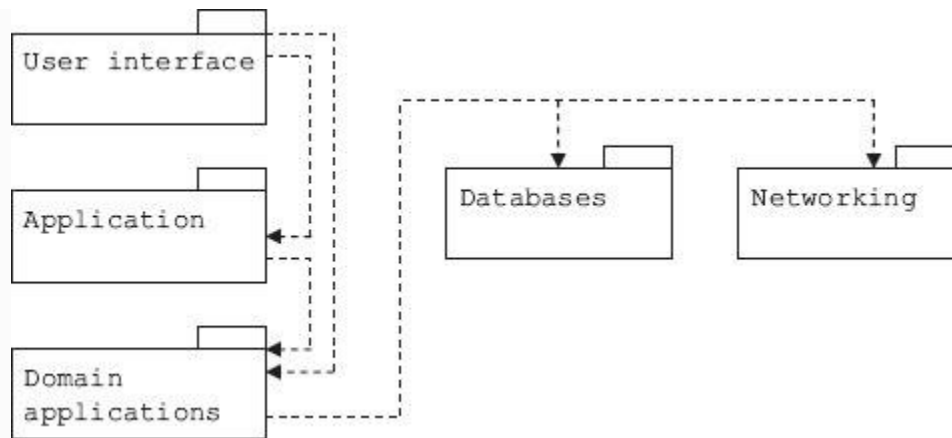
Artefacts is the name given by UML to files. There can be several file types. For example, source code, library file called .h file in C++ notation, executable file. Figure 4.15 makes the concept clear. Here, we show a source called `Student.cpp` and a library called `Student.h` and corresponding object modules with extension `.obj`. The standard artefacts used are the following:

`<<executable>>`, `<<document>>`,  
`<<file>>`, `<<library>>`,  
`<<script>>` and `<<source>>`.



**Figure 4.15** Components (artefacts) in a project

In Figure 4.16, we show how packages are interconnected with dependencies. Dotted arrows show dependencies of a package on other packages.



**Figure 4.16** Packages in a project

## 4.11 Component and Deployment Diagram

**Deployment diagrams** depict the physical configuration of software and hardware. We are aware that in object-oriented technology, class diagrams depict the structure of the application and interactions among them through various relationships such as inheritance.

We are further aware that class contains member functions that contain all the code modules required to achieve functionality.

A component is defined as code module. An application such as online shopping



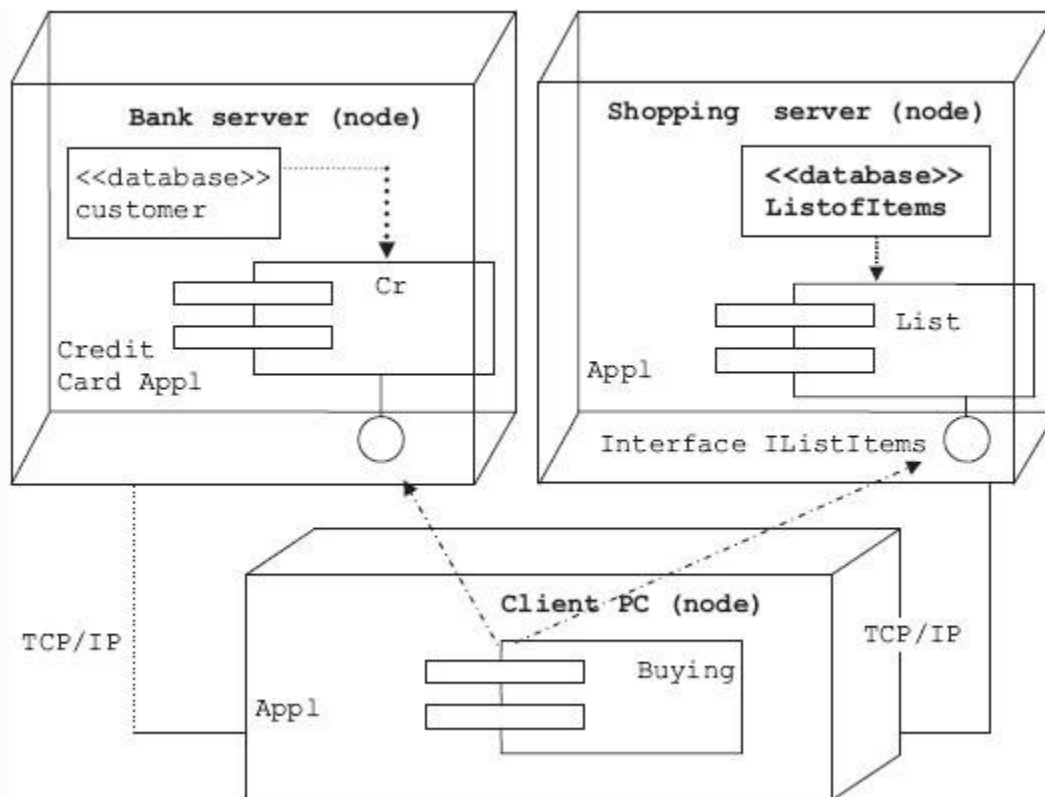
contains several components. An application will have a user interface.

Hardware is a collection of nodes.

Components reside on the nodes.

Remote nodes are interconnected using networking hardware such as transmission control protocol (TCP)/Internet protocol (IP).

Thus, there is a need to group all the above in a single diagram, which is called a component and deployment diagram, which is shown in Figure 4.17.





**Figure 4.17** Component and deployment diagram for online

## 4.12 Summary

1. Modelling consists of structural modelling and behavioural modelling. Structural modelling is also called static modelling. Behavioural modelling is also called dynamic modelling.
2. Class diagrams, object diagrams and package diagrams describe the static characteristics of a system modelling.
3. State of an object is defined in terms of current values held by attributes of an object. It is the result of

behavioural changes of the object till that particular instance.

4. The UML is used as a graphical tool in OOAD methodology to model a software-based system under development. It accommodates all the popular and well-accepted theories of object-oriented technologies such as those of Grady Booch, James Rumbaugh and Ivar Jacobson.
5. Use cases provide a sequence of steps describing the interaction between a system and user. This sequence of steps is called a scenario. Use cases explain the functional requirements of the system under study.
6. Sequence of steps can be represented in a sequence diagram.
7. Sequence diagram with time ordering of events/activities is called timing diagram or communication diagram.
8. We can define architecture as the depiction of objects and their attributes together with the operations performed by these objects to meet user specifications or needs.
9. Communication diagrams were earlier known as collaboration diagrams. These are essentially same as sequence diagrams; but in the case of communication diagrams, time ordering is shown explicitly by numbering the messages.
10. Dynamic model is the collection of all state diagrams that interact with each other with interacting events.
11. States perform activities. An action is immediate operation. An action can also be used to generate other events.
12. Activity diagram shows the flow of activities and their interdependence on one another. The activities concerning each class are shown as rounded rectangles in a separate column called swim lane.

13.  White diamond is used to indicate branching, and guard expressions are enclosed in [].
14.  A thick line is used to indicate fork, i.e., to initiate more than one activity.
15. Package is like a folder. We can store all our classes in this folder directly. If the number of classes is very large, then classes can be grouped into components and components, in turn, into packages.
16. Deployment diagrams depict the physical configuration of software and hardware.
17. A component is defined as a code module. Hardware is a collection of nodes. Components reside on the nodes.

## Exercise Questions

### Objective Questions

#### 1. State diagram specifies

1. State of the object during its lifetime
2. State of the object at current state
3. State of the object in current use case
4. State of the object across all use cases

1. i and iii
2. i and ii
3. i and iii
4. i, ii and iv

#### 2. State of the object is defined by

1. Current attributes
2. Result of all transitions till current state
3. Behaviour of the object
4. Methods

1. i and iii

2. i and ii
3. i and iii
4. i, ii and iv

3. Which of the following statements are true in respect of state diagram?

1. State diagram depicts events
2. State diagram depicts state
3. The transition to next state depends on current state
4. Change in state means change in behaviour

1. i and iii
2. i, ii and iv
3. i and iii
4. i, ii and iv

4. Dynamic model means collection of all state diagrams  
TRUE/FALSE

5. The UML

1. Unifies the theories of Booch, Rumbagh and Ivar Jacobson
2. Unifies development methodologies such as OOSE, OMT and Booch
3. Unifies SASD and OOAD
4. Unifies RUP and development methodologies

1. i and iii
2. i and ii
3. i and iii
4. i, ii and iv

6. UML 2.2 has ----- number of diagrams

1. 12
2. 13
3. 14
4. 15

7. UML 2.2 static view has ----- number of diagrams

1. 6
2. 7
3. 8
4. 9

8. UML 2.2 dynamic view has ----- number of interaction diagrams

1. 4
2. 7
3. 5
4. 3

9. The following is not part of static diagram

1. Class diagram
2. Use case diagram
3. Object diagram
4. Package diagram

1. i
2. ii
3. ii and iv
4. i, ii and iii

10. In use cases, which of the following statements are true?

1. Users are called actors
2. Use cases are sequence of steps
3. Use case contains scenarios
4. Use case means architecture

1. i
2. ii
3. ii and iv
4. i, ii, iii and iv

11. Which of the following is true in respect of sequence diagram?

1. Actor is represented by a stick figure
2. Objects are represented by oval
3. Vertical line represents activity
4. Horizontal arrow represents messages

1. i and ii
2. ii
3. i, iii and iv
4. i, ii, iii and iv

12. Which of the following is true in respect of communication diagram?

1. They are also called collaboration diagrams.
2. They are same as sequence diagrams.
3. They are sequence diagrams with time ordering of activities.
4. They concentrate on object role rather than on time at which communication is sent.

1. i and ii
2. ii
3. i, iii and iv
4. i, ii, iii and iv

13. Which of the following is true in respect of activity diagram?

1. Activity diagram depicts object undergoing a process.
2. Activity diagram is flow of activities in a single process.
3. It is flow of activities and their interdependence on one another.
4. They concentrate on object role rather than on the time when communication is sent.

1. i and ii
2. ii
3. i, iii and iv
4. i, ii and iii

14. Which of the following is true in respect of activity diagram?

1. White diamond means branching
2. Thick line indicates fork
3. Fork means initiation of one activity
4. [] are for guard expression

1. i, ii and iv
2. ii
3. i, iii and iv
4. i, ii and iii

15. Which of the following is true in respect of packages and components?

1. Packages contain components
2. Components contain classes
3. Files in UML are called artefacts
4. Files, library and script are part of artefacts

1. i, ii and iv

2. ii
3. i, ii, iii and iv
4. i, ii and iii

16. Deployment diagram includes configuration

1. Software
2. Software and hardware
3. Hardware
4. Server details

**Short-answer Questions**

17. Explain the term “Unified” in Unified Modelling Language.
18. Define architecture and domain.
19. List the names of diagrams that are part of static view of UML.
20. List the names of diagrams that are part of dynamic view of UML
21. Explain composite structure diagram and profile diagram of UML.
22. Distinguish sequence and communication diagrams.
23. Explain the terms event and state.
24. Explain the terms activity, action and operations.
25. Explain nested state diagrams.
26. What is dynamic modelling?
27. Explain the terms packages and components.

**Long-answer Questions**

28. Explain in brief UML static and dynamic diagrams.
29. Explain use cases with an example.
30. Explain the terms architecture and domain
31. What are sequence diagrams? Explain the concept with an example.
32. What are communication diagrams? Explain the concept with an example.
33. Explain event and state.
34. Explain nested state diagrams with an example.
35. What are the various artefacts in a project?



### **Assignment Questions**

36. Write history and background for the development of UML as an OOAD tool.
37. Explain dynamic modelling. What aspects of a project can be understood better with this model?
38. Use case represents architecture. Justify the statement.
39. Develop use case diagram for ATM banking operation.
40. Develop a state chart diagram for online Internet shopping.
41. Draw a state diagram for online checking of the results declared by university.
42. Develop the activity and deployment diagram for the Problem 6.

### **Solutions to Objective Questions**

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



# 5

## Analysis and Design Methodologies

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to understand and use techniques and methods regarding*

- Various methodologies of system development.
- Complete life cycle development phases in a software development project.
- Structured analysis and design, including function modelling and data modelling.
- Data flow diagrams and data dictionaries and entity relation.

## 5.1 Introduction

Detailed and in-depth analysis of issues involved in a system under development or an existing system would lead to better understanding and thus better designed systems meeting user requirements in letter and spirit. If we can model analysis and design phases using the latest tools like UML, software development will be in consonance with pacifications and thus greatly enhance the productivity of developers and designers.

In this chapter, we cover details like software development life cycle and design methodologies that include in-depth coverage of structured analysis and design as well as object-oriented analysis and design. We present theoretical foundation as well as case studies so that you can grasp the concepts. We have also included elaborative examples and assignments at the end of the chapter. There are programmers and programmers. But there are only a handful of competent designers. Thus, this chapter is

important for budding analysts and engineers.

## 5.2 Stages and Methodologies for Systems Development

A complex project involves clearly two major stages: analysis and design phases. The project development team goes through the following stages in project implementation spread over analysis and design phases:

- **Inception:** This initial stage is used to elicit information from the users regarding the problem and estimation of project costs and the formation of a senior project team.
- **Elaboration:** This is under the analysis phase and is used to elaborate the analysis carried out during the initial stages. Estimations and conceptual understanding of the project is undertaken. Tools like UML and CASE are employed at this stage.
- **Construction:** This stage falls under the design stage wherein design methodologies like structured design (SD) or object-oriented design (OOD) are resorted. This section also includes coding the system as per design.
- **Coding:** This stage is about translating the design into coding using a structural language like C or object-oriented language like C++ or Java based on the methodology of development.
- **Testing and Delivery and Deployment:** The project developed undergoes elaborate testing to check if the requirements are met. The testing is also carried out

after delivery of the project at the users' site. Testing methods depend on the methodology chosen like structural analysis and design (SASD) or object-oriented analysis and design (OOAD).

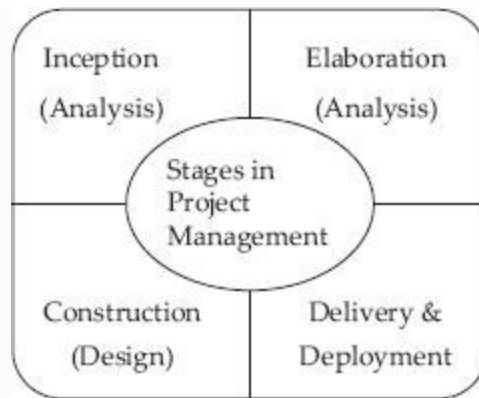
### *5.2.1 Methodologies for Software Development*

There are many methodologies for the development of information systems: systems development life cycle (SDLC) and OOD are widely used and popular design methodologies.

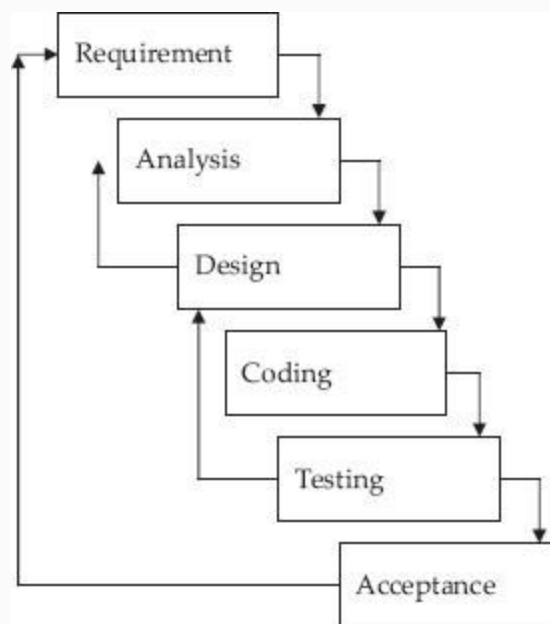
### *5.2.2 Systems Development Life Cycle (SDLC)*

SDLC is closely linked to structured system analysis and design (SASD). SDLC is also referred to as waterfall model.

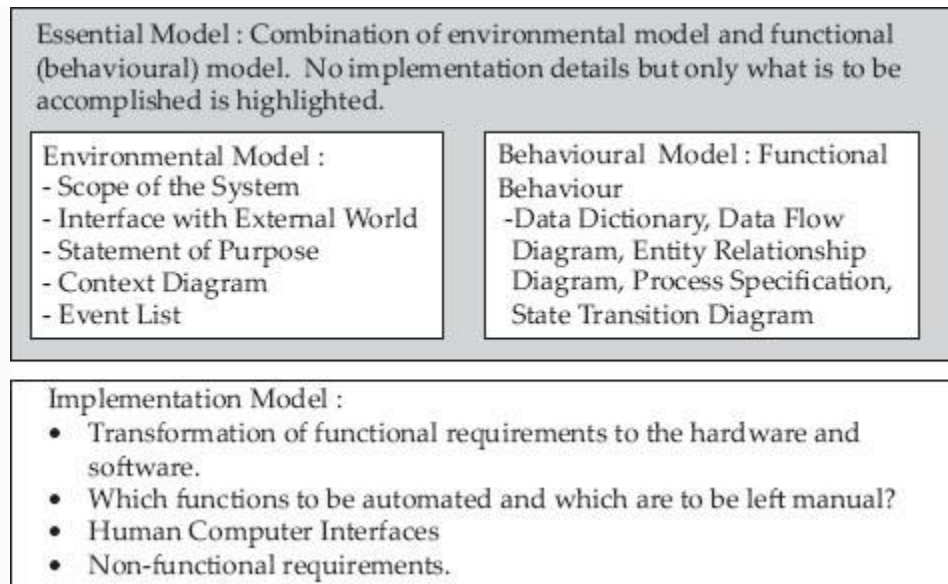
It is called waterfall model because it looks like one. It is a model that describes the steps taken in the development of information systems. This concept has been introduced by Mr Horner in 1993.



**Figure 5.1** Stages in project



**Figure 5.2a** SDLC – waterfall model



**Figure 5.2b** Structured analysis and design – main elements

The series of action and steps to be taken and deliverables at each step are shown below:

- **Problem definition:** The user would have prepared a problem narrative. On receiving a request from the user for systems development, an investigation is conducted to state the problem to be solved. **Deliverables:** Problem statement.
- **Feasibility study:** To define the scope and objective of software project study and also establish alternative solutions available. **Deliverables:** Feasibility report.
- **Systems analysis phase:** Analysis of existing system. How and what it does. **Deliverables:** Specifications of the present system.



- **Systems design phase:** Study of specifications and preparation of a specifications document showing what else needs to be done to obviate the shortcomings of the existing system. **Deliverables:** Specifications of the proposed system.
- **Systems design:** Development of code, user manuals. **Deliverables:** Programs, their documentation and user manuals.
- **System testing and evaluation:** Testing, verification and validation of the system against specifications. **Deliverables:** Test and evaluation results, and the system ready to be delivered to the user/client.

The advantages of the waterfall model are as follows:

- Clearly defined deliverables at the end of each phase, so that the client can take decisions on continuing the project.
- Resources can be committed incrementally and not all at once.
- Problems can be detected early in the developmental stage.
- Detailed stagewise documentation.

The disadvantages of the waterfall model are as follows:

- The project needs to be committed. It is not an incremental model.
- Problems have to be detected early. Later stage detection means huge costs.

### 5.3 Structured Analysis and Design (SASD)

SASD is a methodology adopted to analyse the business model and convert it into specifications, which in turn will be modified into computer programs, hardware and/or manual procedures. SASD as a methodology was developed in the late 1970s by DeMarco, Yourdon and Constantine after the emergence of structured programming. The purpose of SASD is to develop a useful, high-quality information system that will meet the needs of the end user. CASE tools are graphical tools to implement SASD techniques and have now become industry standard for developing SASD models.

#### **Elements of Structural Analysis and Design:**

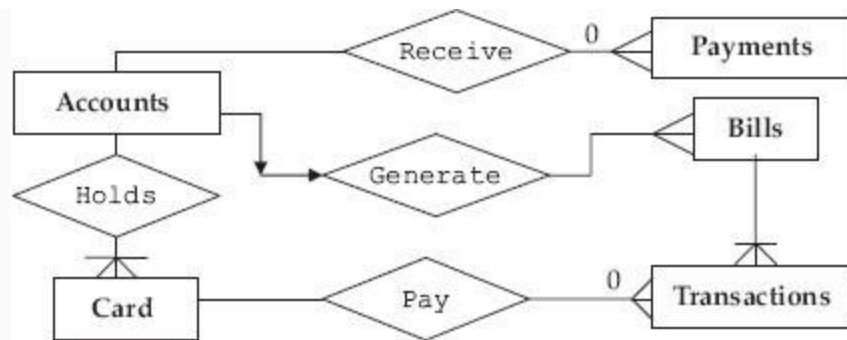
The steps involved in SASD are as follows:

##### **STEP 0: Statement of Purpose**

This is a clear and concise statement highlighting the purpose of the system. It is meant for higher management as a concept document and cannot be used for development of the system.

**Example of statement of purpose** – The purpose of the “Agri Card System” is to provide a methodology for the bank to extend credit to the farmer. The system will handle details of credit application, loan management, billing, recovery and management reporting. The steps involved are:

**STEP 1:** Draw the **context diagram** to define the scope of the project. This diagram is like a block diagram, showing the system as a whole with set of inputs and outputs and a transformation system. The objective of a system context diagram is to focus attention on external factors and events that should be considered in developing a complete set of system requirements and constraints. In Figure 5.7, we have provided a context diagram for Student Smart Card Application.



**Figure 5.7** ERD for credit card system

**STEP 2:** This is documentation showing how the existing system works. It is accomplished by drawing the physical data flow diagrams (DFDs) of the existing system that specify its current implementation. The key questions raised are:

- Who performs what tasks?
- How and when are they performed and what is their frequency of performance?
- How and where is the data stored?

**STEP 3:** This stage is for documentation of the proposed system. It is about what the proposed system will achieve. The system designer will study the lacunae of existing

system and propose the modified system in the form of logical DFDs.

**STEP 4:** At this stage, logical DFDs at STEP 3 are converted to physical DFDs by examining which implementation meets the criteria. Who will perform the various tasks?

- Who performs what tasks?
- How and when are they performed and what is their frequency of performance?
- How and where is the data stored?

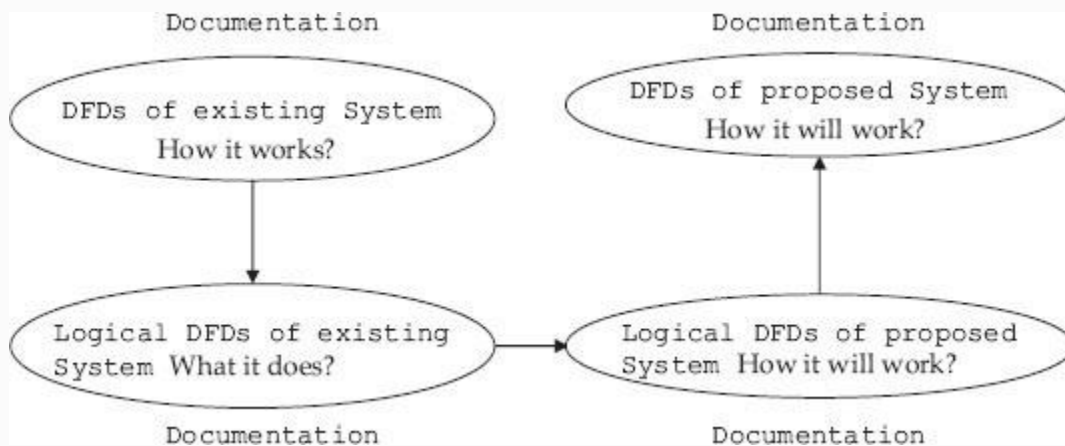
### *5.3.1 Conceptual Design – Functional Modelling and Data Modelling*

Conceptual design consists of data modelling and functional Modelling. Data modelling is a technique to organize the data and functional modelling (also called process modelling) is a technique to document systems process, inputs and outputs. Conceptual design phase throws up answers to questions such as

- What are the processes making up the proposed system?
- What data is used in each process?
- What are the inputs?
- What are the outputs?

### 5.3.2 Modular Design

The proposed system is broken into small independent subsystems called modules. These modules are designed and coded separately and are later combined with other modules to provide complete functionality of the proposed system. We will adopt a top-down hierarchical model in which higher-level modules will have larger scope and lower-level modules will have smaller scope. The design considerations for breaking in to modules are:



**Figure 5.3** DFDs of existing and proposed system

- The size of the module as indicated by lines of code should be small.
- The modules must be sharable so as to avoid duplication.

### *5.3.3 Analysis and Design Techniques and Tools*

The analysis part will involve converting business model into data flow and control flow. This is achieved through the use of DFDs. Data dictionaries are essential to describe the data and control flows. DFDs are graphical representations of the functional decomposition of the main process involved. Process specifications and specifying operations are helpful in decomposing into cohesive submodules with coupling between functions leading to structured data. These subdivided systems in turn retain constraints and specification of originally designed system. SASD technique attempts to solve the complex problem by dividing large, complex problems into smaller, more easily handled ones. The technique can be called the “divide and conquer method.” The approach

adopted can also be called the “top-down approach.” The functional decomposition of the structured method describes the process without delineating system behaviour and dictates system structure in the form of required functions. The method identifies inputs and outputs as related to the activities. The result of structured analysis is a set of related graphical diagrams, process descriptions and data definitions. They describe the transformations that need to take place and the data required to meet a system's functional requirements.

#### *5.3.4 Context Diagrams*

This diagram is like a block diagram, showing the system as a whole with a set of inputs and outputs and a transformation system. The objective of a system context diagram is to focus attention on external factors and events that should be considered in developing a complete set of system requirements and constraints. We have provided an example of context diagram along with a case study in [Figure 5.7](#).



## **Purpose**

- Highlights the boundary between the system and the outside world
- Highlights the people, organizations and outside systems that interact with the system under development
- Special case of the data flow diagram

## **Context Diagram – Notation**

- Process – Represents the proposed system → Flow. Data in/out
- Terminator – Represents the external entities

### *5.3.5 Event List*

The purpose of event list is to prepare a list of external events/activities to which system under design should respond. This is similar to use cases of OOAD methodology. The events can be classified as

- Event triggered by incoming data
- Internally generated event
- External unpredictable event

The examples for event list are:

- Student completes registration process

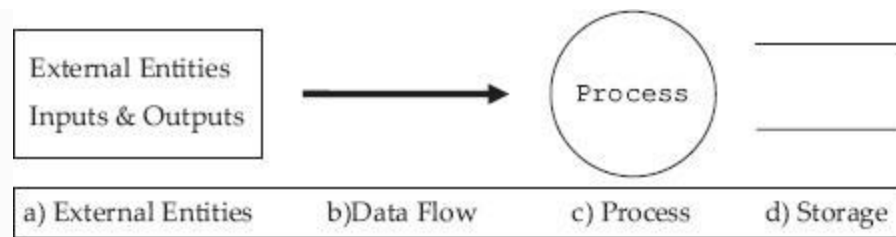
- Customer pays credit card bill

### *5.3.6 Data Flow Diagrams*

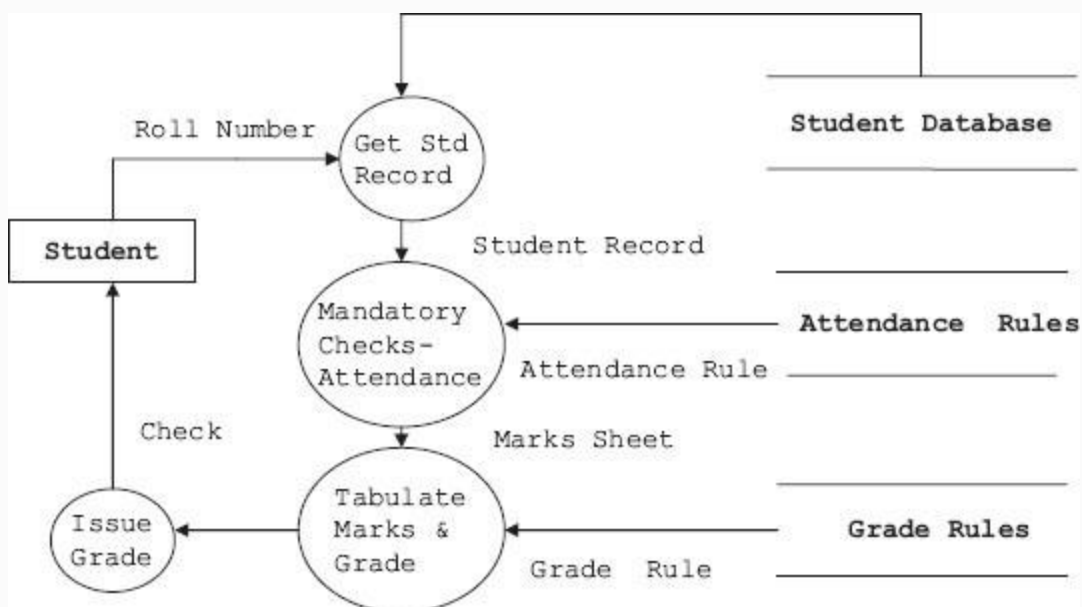
Data flow diagrams (DFDs) also called data flow graphs are used in the analysis phase as they are useful in understanding a system and can be effectively used during analysis. A DFD views function as data flows through the system. DFDs highlight the data transformation from input stage to output stage through various processes.

#### **Example 5.1: DFD for Issue of Grade Card for Student**

We will show the DFD for preparing students' grade sheet. We will call this system as GradePrep system, as shown in Figure 5.5. In GradePrep, inputs and outputs are:



**Figure 5.4** Symbols in DFD



**Figure 5.5** DFD for issue of grade sheet to a student

- Students personal data base which stores basic information about student such as name, roll number, marks obtained in subjects and attendance details.

- Grade Rule database would store information about criteria for calculating grades and Attendance Rule database contains rules regarding mandatory attendance required to declare the result, etc.
- Students' grade sheet is output.

### 5.3.7 Data Dictionary

A data dictionary or ***database dictionary*** is a file that defines the database. A database dictionary contains

- List of all files and tables in the database.
- The number of records in each file.
- Names and types of each data field.

Data dictionary only maintains information to manage data but holds no actual data. Database management system cannot function without data dictionary. We can call it meta data about data but not actual data. In the DFD shown above, we have used terms like student database, attendance rules, grade rule, etc. But what exactly is the structure of data in these storage spaces. Data dictionary defines data structure in these storage spaces:

---

- Student Record = roll number + Name  
+ course + Semester + marks1 + marks2 +

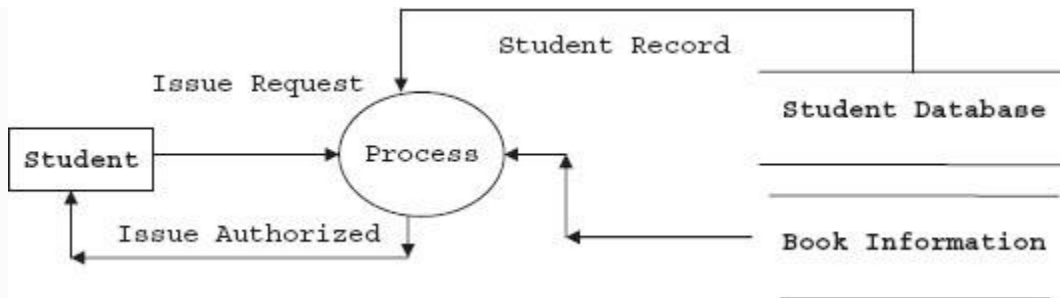
marks3 + marks 4 + marks 5 + Attendance  
percentage

- Attendance rule: if (attendance > 75) eligible = 1 else eligible = 0
  - Marks Sheet: roll number + Name + course + Semester + marks1 + marks2 + marks3 + marks 4 + marks 5 + Total Marks + Grade
  - Roll Number: digit + digit + digit + digit + digit (As many digits as there are in roll number)
  - Grade: String "A" or "B" or "C" or "D," etc.
- 

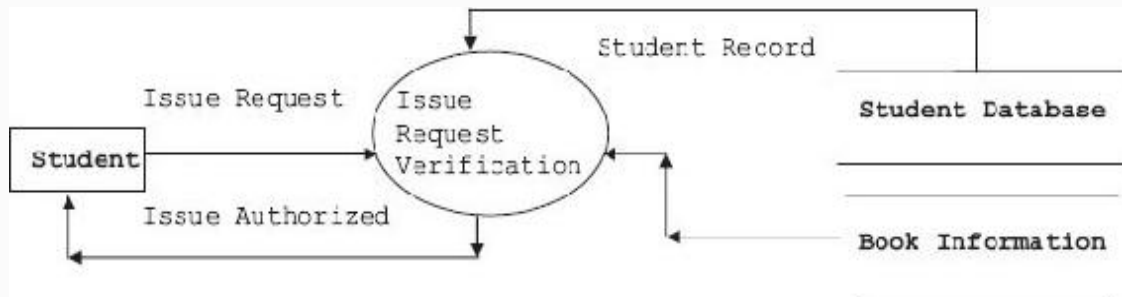
### *5.3.8 Multilevel DFDs*

The DFD and data dictionary for the DFD are ready; we can expand the DFD into second and subsequent levels to understand the process involved better. For this study, let us consider the library information system (LIS) of a university or a college. Student selects a book for issue based on availability status checked online. The library staff has to process the request for issue, verify and validate the request, and process the request. Finally, the student is issued the book with issue authorization

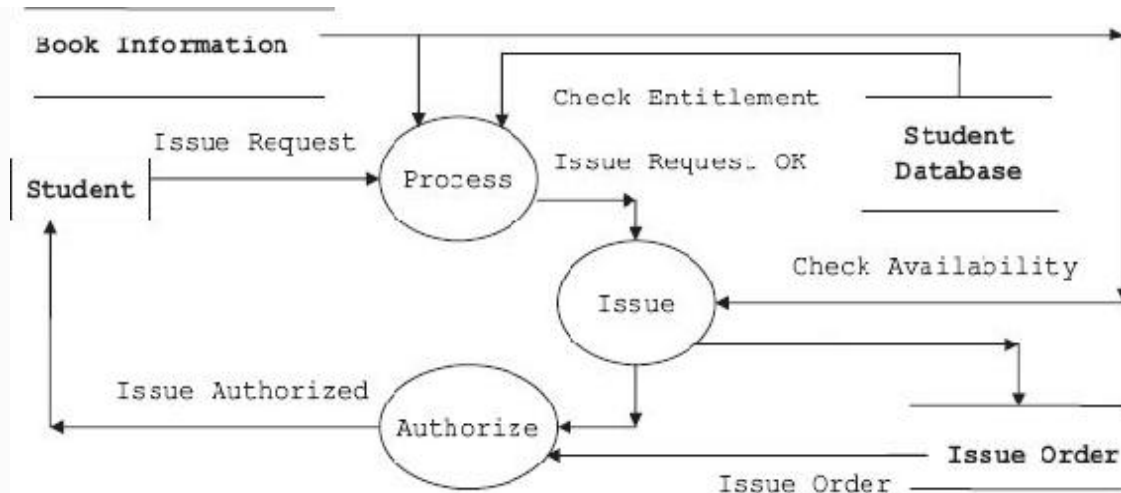
note and his account modified accordingly. Note that we have selected LIS for this case study because you are familiar with the workings of a library. We can easily replace the model with any other model like purchase–order system, etc. In Figure 5.6a, b, c, we have shown how a DFD can be expanded to gain insight into how an issue section of a library works. First, incoming issue requests are checked for correct book titles, authors' names and other information. Student's entitlement is also checked at this stage by checking the students' database. This process is elaborated at DFD level 2. Finally, the process of issue is elaborated at DFD level 3. We show the set of DFDs drawn for issue section of LIS in Figure 5.6a, b, c.



**Figure 5.6a** First-level DFD for issue of a book



**Figure 5.6b** Second-level DFD – issue request verification



**Figure 5.6c** Issue request processing system

## Example 5.2: Multilevel DFD for Issue – Return System of Library Information System

**Level 1: General overall concept diagram**

**Level 2:** Expand the process of level 1. At this level, we will check the request for issue and verify *if the student is entitled to draw*



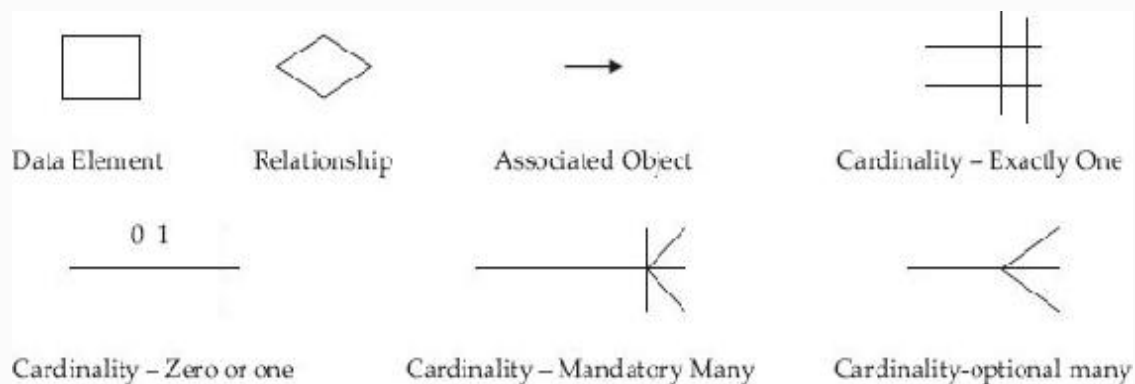
*the book and further if the book can be issued.*

## **Level 3: Expanding issue request processing system**

### *5.3.9 Entity Relationship Diagram*

Entity is like an object. It is a thing present in the system under development. It is a graphical representation of the data layout of a system. It defines data elements and their interrelationships in the system.

#### Entity Relationship Diagram — Notation

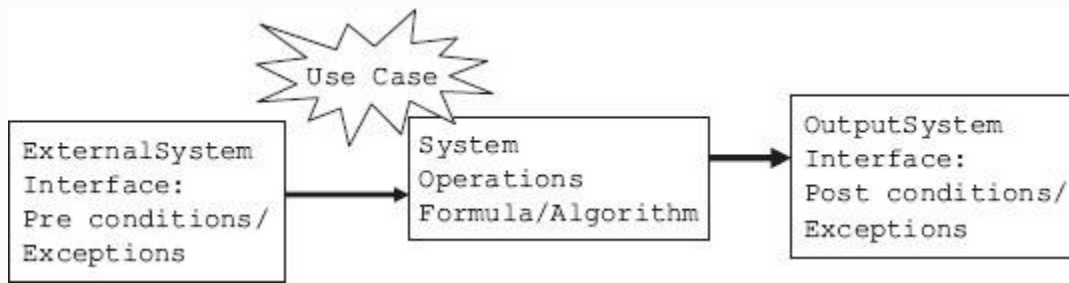


Let us study the ERD for the credit card system of a bank. The entities in this system are payments, bills, transactions, accounts, transaction elements. Relationships are:

- User pays for a transaction using card
- Accounts section hold card information
- Accounts generate bills
- Bills contain transactions
- Accounts receive payments

### *5.3.10 Process Specifications*

Shows process details, which are not shown in a DFD. Input, output and algorithm of a module in the DFD are included in process specifications. Observe at each operation and define the transformation of inputs to outputs in the system functional model and the functional model. We desire to define the transformation in terms of computation, algorithms and formulas. Specifying operations is like specifying the attributes. Constraints will help the designer to develop the system as per specifications by ensuring one-to-one mapping of functionality, i.e. operations. Specifying operations is usually carried out in graphical modelling so that they define underlying constraints with precision. The net result is better designed systems. So when we intend to specify the operation, what are the areas we need to specify? They are:



**Figure 5.8** System view – inputs – system – output

1. **Internal behaviour:** This can be specified both by algorithmic and non-algorithmic methods. The non-algorithmic approach refers to using decision tables and setting up of pre- and post-operation values. A transformation could be described as follows:

1. A table showing the mapping from one object value to another
2. A formula or equations showing the relationship between an input object value and an output object value
3. A text description of the transformation

2. **Interface with external entities:** Interfaces specify the inputs to the system. Specifications can be used as a means to check if functionality is meeting the requirements. In Figure 4.21, we presented the functional model as a system view with inputs, outputs and system transformations.

Operations can be classified as system operations and functional operations. System operations are those that affect the whole system. For example, withdrawal of cash from ATM would affect the entire system.

Functional operations are local in nature and affect the attributes called by that particular module.

### 3. Procedure for Specifying the System Operations and Function Operations.

1. Identify all system/function operations
2. For each system operation, state the system operation name, input objects (parameters), transformation and output objects (return values). In addition, state the system operation preconditions, post-conditions and exceptions.
3. Prepare a system/function operation table, as shown below:

Operation	Input	PreCond/Excep	Transformation	PostCond/Excep	R/T
-----------	-------	---------------	----------------	----------------	-----

#### 5.3.11 Specifying Constraints

Functional model can best be described by a graphical representation. Graphical representation is nothing but functional decomposition described in earlier sections, wherein two distinct paths, namely, control path and data path, flow from top to bottom covering all decomposed modules.

Therefore, all functionality constraints applied at the topmost level must also be applied to decomposed lower levels. For example, if reliability is a constraint of the topmost module, then it must also necessarily be a constraint in succeeding modules. Constraints and their implications are discussed next.

- **Performance:** Performance criteria are defined at the topmost layers. Performance dictates the operations. Operations, in turn, decide the functionality of the module. So when a top module is decomposed, care must be taken to ensure that operations at each submodule are as per specifications.
- **Reliability:** In software, reliability of function modules depend on the way the reliability aspect is distributed among the subdivided function modules and the way user interacts with these modules. There should be match between user operational profile and functional profile.
- **Security:** Security issues crop up because the user at operational level is able to exploit some functionality that is not part of the original top-level specification.

This loophole might have been created inadvertently at lower functional level. The criteria to be considered to plug these loopholes are described below:

- **Access Control:** Subdivision of functionality must ensure that only certain classes or users are able to initiate and use the functionality with authorization codes.
- **Integrity of Data:** The data is primacy and no unauthorized access can be allowed at submodules. A suitable encryption technique can solve these problems.
- **Control of User Behaviour:** Normal behaviour of a user of a function module must be specified and any deviations from this normal and designed process must be recognized by the module and further exploitation of function module must be denied.

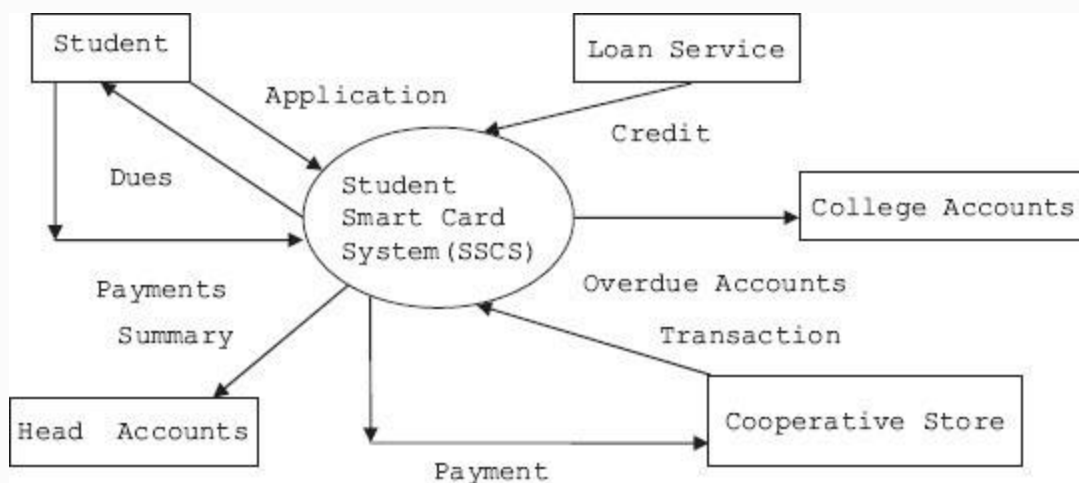
- **Availability:** We are aware that user operations are distributed over submodules and users must be guaranteed that they will be able to operate these modules with reliability and with intended specifications at all times. We call this aspect availability of function modules. Availability means that we must be able to monitor the working of functionality as per specifications and it should be measurable.
- **Maintainability:** The success of designed systems depends on the ability of modules to offer operations as per specifications. This means the mapping between functionality and operations must be verifiable and assured. We term this aspect as maintainability of the system.
- **Vital or Non-vital Functionality:** It is advisable to divide the functionality to vital or otherwise and concentrate on vital functionalities so that operations and functionality can be mapped and specifications can be ensured.
- **Compatible with hardware and other functional module:** Decomposition of modules must ensure that operation at each level is compatible with hardware specifications and other function modules.

## 5.4 Case Study on Structured Analysis and Design

### *5.4.1 Example of Statement of Purpose*

The purpose of the student smart card system (SSCS) is to provide a means for the college administration to extend all due

facilities to students. The system will handle details of fees paid, details of loan applications, issue of library books and payment of fines if any, attendance recording and management reporting. Information about transactions should be available to the corporate accounting system and also to the academic administration of the college.



**Figure 5.9** Context diagram for student smart card system

### *5.4.2 Context Diagram*

Context diagrams depict the boundary between the system and the outside world,

and highlights the people, organizations, and outside systems that interact with the system under development. The context diagram is presented in Figure 5.7.

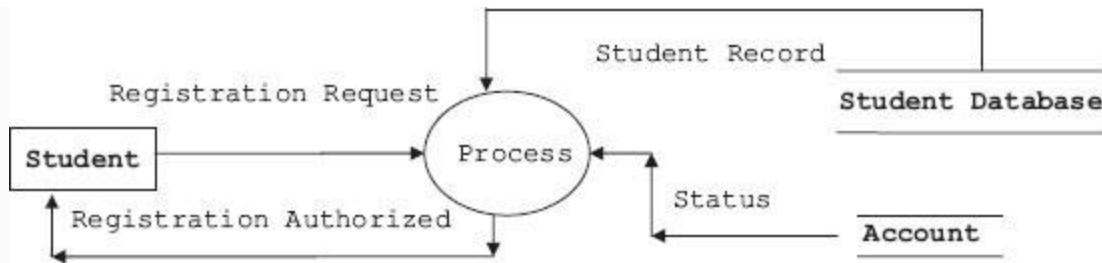
### *5.4.3 Event List*

- Student applies for a smart card
- SSCS informs student his dues
- Student pays his dues
- Student makes a transaction with coop stores
- Student is offered a loan
- Student has been given a credit

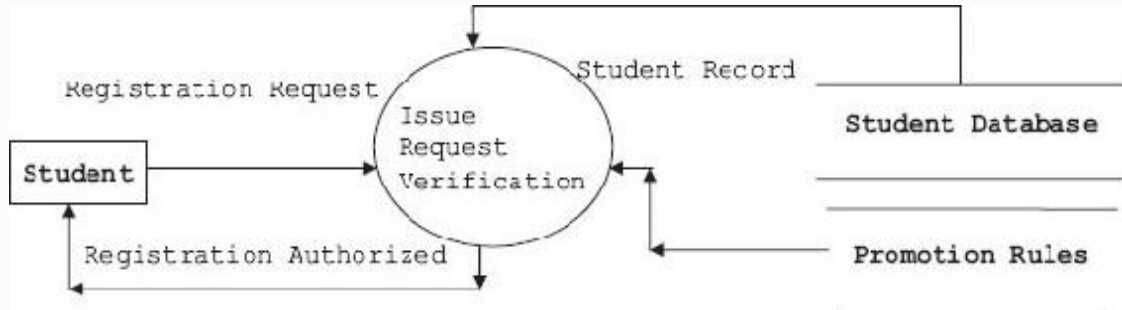
### *5.4.4 Data Flow Diagrams*

Student requests for registration. The SSCS system checks status of tuition fees paid from accounts section and also obtains students' records from students' database and processes the request. If found eligible, the system issues authorization for registration process.

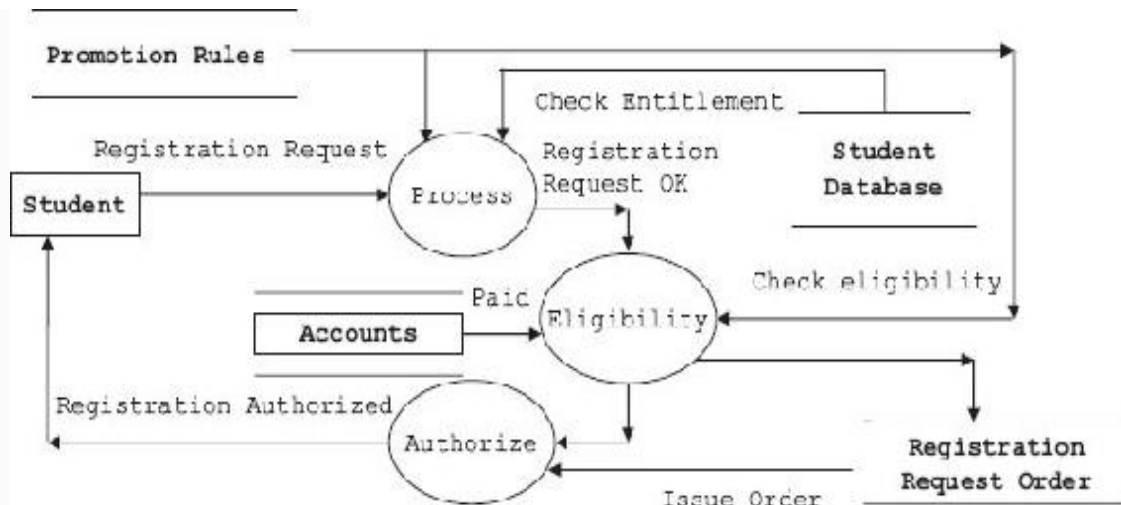




**Figure 5.10a** DFD for registration process – SSCS system



**Figure 5.10b** DFD for registration request verification



**Figure 5.10c** Issue request processing system

### 5.4.5 Data Dictionary

Data dictionary only maintains information to manage data but holds no actual data. Database management system cannot function without data dictionary. We can call it meta data about data but not actual data. In the DFD shown above, we have used terms like student database, promotion rules, account database, etc. But what exactly is the structure of data in these storage spaces. Data dictionary defines data structure in these storage spaces:

---

Student Record = roll number + Name +  
course + Semester + No of Back lag  
papers weightage

Promotion rule: if (backlog paper  
weightage < 25) eligible = 1 else  
eligible = 0

Reg. Authorization Sheet: roll number  
+ Name + Semester + courses registered

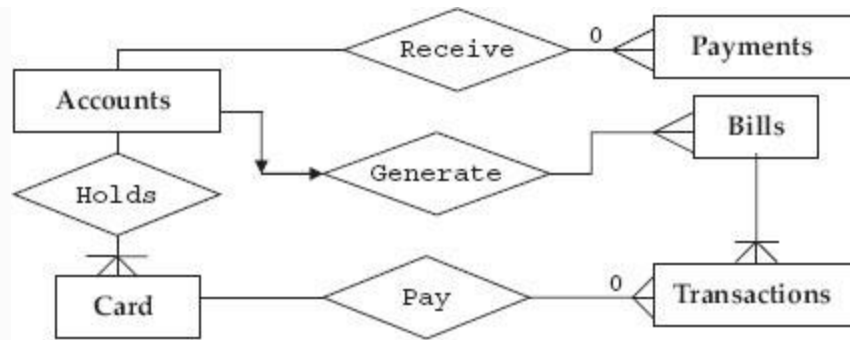
Roll Number: digit + digit + digit +  
digit + digit (As many digits as there  
are in roll number)

Student Smart Card: Card Number =  
Uniquely identifies a card

Format = digit + digit + digit +  
digit + digit + Hyphen + alphabet

Range = 50595 0000 0000 0000 to 50595  
9999 9999 9999

---



**Figure 5.11** ERD diagram for smart card payment system

### 5.4.6 Entity Relationship Diagrams

Entities in the SSCS for transactions and payments by a student are: payments, bills, transactions and accounts. Relationships are:

- Student pays for a transaction using smart card
- Accounts section hold smart card information
- Accounts generate bills
- Bills contain transactions
- Accounts receive payments

### 5.4.7 Process Specifications – SSCS

- For payments, entity process specifications are:
- Read account
- Read amount

- Add amount to account's credit
- Add amount to account's balance
- Update payment as applied

We leave writing specifications for balance entities for transactions, card, accounts, etc. as an exercise for the reader.

### *5.4.8 Structured Analysis and Design – Advantages and Disadvantages*

#### **Advantages**

- It has clearly distinguishable phases, hence amenable to project management.
- DFDs, ERDs, etc., are easy to understand graphical tools.
- Widely used in industry and mature technique.
- Function/process oriented. It follows natural thinking process.
- Modular and hence flexible.

#### **Disadvantages**

- Only functional requirements are met. Ignores non-functional requirements.
- User–analyst interactions cannot be automated. It is manual and non-iterative.
- Communications with user after requirement analysis is non-existent.
- Hard to decide when to stop decomposing.

## Where to use structure analysis and design

- SASD is best suited for very well-known and standard domains.
- Areas where SRS is specified.
- In transaction processing systems.
- When good amount of development time is available.

### 5.5 Object-oriented Software Analysis and Design (OOAD)

OOAD as it is popularly called is structural analysis and design and certain object-oriented features built over it. Accordingly, we start with user specifications and add the following object-oriented features:

- List of objects drawn out of grammatical parsing of narrative statement. Data content of nouns or entities from DFDs are of interest to us.
- List the system behaviour from verbs.
- Service provided by the object to other objects.
- Expand the objects and the relationships with other objects.

#### *5.5.1 Different Models for Object Analysis*

In OOAD methodology, interaction with user is very heavy. It does not end with drawl of specifications and user signs off the

document. OOAD methodologies depend on various methodologies propounded by Rumbaugh, Grady Booch, Coad–Yourdon and Shlaer–Mellor. We will review these systems in the next section. The requirements keep changing during the development period of a complex project and there is a requirement to effect these changes and many a times this aspect causes design changes from step 1. OOAD works with the assumption that specifications will change and hence follows an iterative development pattern. Each iterative step either adds a new step or modifies the existing step.

#### **5.5.1.1 The Rumbaugh Method**

Rumbaugh is associated with the development of OOAD methodology that has a well-defined analysis as well as a design model. The analysis wing of the Rumbaugh model deals with object modelling technique (OMT). It further includes a model to study the dynamic behaviour of the system under development. The Rumbaugh model also includes functional modelling. Rumbaugh's

model is similar to the SASD techniques discussed so far, with the additions for the object model, including definitions of classes along with their member data and functions and their interactions with other classes. Dynamic modelling includes state transition diagrams that show how an object changes from one state to another as an event occurs. Functional specifications are taken care by DFDs.

#### **5.5.1.2 The Booch Method**

Booch's methodology includes requirement analysis phase that is similar to the SASD practice. In addition, Booch's methodology includes domain analysis phase as part of the analysis phase. Booch's methodology enjoys strong design techniques and is divided into four parts:

- Part 1: Logical structure design where the class hierarchies are defined
- Part 2: Physical structure describing objects methods
- Part 3: Dynamic modelling showing state transitions and analysis of object transitions
- Part 4: Functional specifications

#### **5.5.1.3 The Coad–Yourdon Method**



Analysis phase is called SOSAS, meaning five steps in analysis and design phases:

- S stands for subject. It means DFDs.
- O for objects. Identify objects and class hierarchies.
- S for structure. It is divided into inheritance structures and composition structures. The classification is based on which of the two methods of extending the class, i.e. inheritance or composition is used in design.
- Attributes.
- Services offered by this class to others.

Design phase: It consists of four parts:

- Problem domain. Contains classes that belong to the problem domain.
- Human–computer interaction.
- Function management. Systemwide classes responsible for functions and management are identified.
- Data management: attributes of the system.

#### **5.5.1.4 The Shlaer–Mellor Method**

The model has three parts in system analysis and design:

- Information model comprises objects, member data, its relations with other objects.
- State model comprises different states of the objects and changes that can occur as objects do a transition.
- Process model takes care of functional specification.

In our book, we follow the best of all methodologies suggested above.

The first activity to be performed in OOAD is requirement gathering. A series of meetings with users gives us a narrative of the problem involved. Users based on the experience gained in operation of a manual system of shortcomings in an existing system would have prepared problem statements or using the tools available may have use cases. Object-oriented classes support the object-oriented principles of abstraction, encapsulation, polymorphism and reusability. **Classes are specifications for objects.** Derived from the use cases or problem statement or narrative, classes provide an abstraction of the requirements and provide the internal view of the application. The steps we would follow in OOAD are as follows.

### *5.5.2 Identifying Classes*

Identifying classes can be challenging. Poorly chosen classes can complicate the applications logical structure, reduce reusability and hinder maintenance.

- Once narrative is available, we do a grammatical parsing to identify nouns, verbs, etc. Make an initial list of nouns. We call this list as **candidate classes**.
- **Candidate Classes:** Initial set of classes from which the actual set will emerge. Candidate classes can be identified by several methods. Two of the methods are shown below:
  - Grammatical parsing of narrative or use cases or problem description: Identify the nouns and noun phrases, verbs and adjectives.
  - Class responsibility collaboration (CRC) cards. These are a collection of cards, wherein class name, its responsibilities (functionality) and collaborations (service to other classes) are listed.
- Analyse the candidate classes. Look for collaborating classes. How does one class relate to another class? If it is collaborating, then retain the class else discard it.
- Follow the principle of selection of final classes suggested by Coad and Yourdon to select a final list of classes. We call this set as design classes.
  1. Does the class have retained information?
  2. Is the service provided by the class needed by other classes?
  3. Does the class have multiple attributes?
  4. Does the class have common attributes that are useful to others?
  5. Does the class have common operations that are useful to others?
  6. Does the class come under essential category and is required by other classes?

### *5.5.3 Identifying Attributes*

Attributes define the characteristics of the class. When looking for attributes in the design model, look for these types:

1. Look for descriptive attributes.
2. What characteristics distinguish this class from other classes?
3. Names are used to uniquely identify the objects. What characteristics uniquely identify this object?
4. How is a particular object linked to other objects?

### *5.5.4 Specifying Operations*

For specifying the class operations, identify the operations each class performs. This list stems from the responsibility of the class. Operations specify the behaviour of the class. The operations can be classified as

- Modifier operations. These operations modify the attributes.
- Checks status of operations through checking a few attributes' values.
- Transformation operation runs an algorithm or formula.
- Operation to check occurrence of an event.

Once operations and attributes are decided, we can finalize the definition of class. Let us use the college administration system for demonstration of OOAD principles. The design classes identified from the narrative are: professor, student, course, etc.

### *5.5.5 Work Out Associations*

Associations are the key to identifying cohesive classes. Classes are associated with, or related to, other classes. Relations between classes occur when a class has some service to offer to other classes or uses services offered by other classes. **A relationship is an association between classes.**

## **How to identify the relationships?**

- Start with a core class that interacts with many other classes.
- Ask questions like: Who is interested in this class? Why is this class required?
- Who uses services provided by this class?

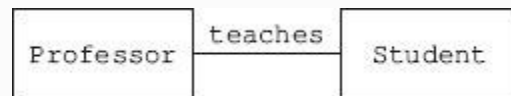
### **5.5.5.1 Start with Main Associations**

In our case, we will start with professor and student. Association, as we have learnt earlier, is a solid line that connects two classes.



**Figure 5.12a** Professor is associated with student

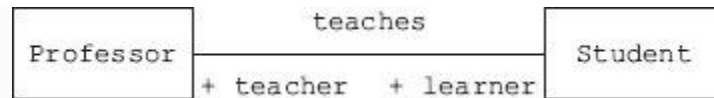
#### 5.5.5.2 Many a Times Naming the Association Clarifies the Relationship



**Figure 5.12b** Naming the association

#### 5.5.5.3 Highlight the Roles of the Classes in the Association

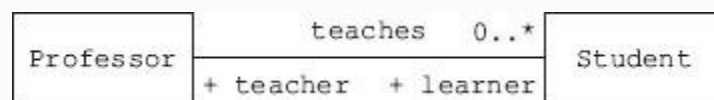
Giving a role is optional. But when given, it must be a noun for a class. For example, we give a role of teacher to a professor and learner to a student.



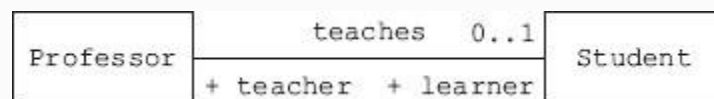
**Figure 5.12c** Roles in an association

#### 5.5.5.4 Introduce Multiplicity Factor

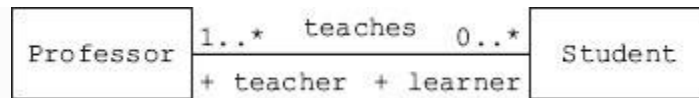
A role can have multiplicity relationship. Multiplicity indicators can be conditional or unconditional. For example:



**Figure 5.12d** A professor can teach 0 to many students



**Figure 5.12e** A student can teach 0 to 1 student

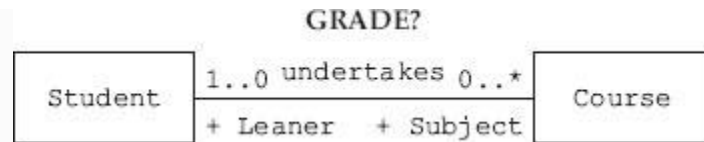


**Figure 5.12f** A professor can have 0 to many students. But a student must have at least one professor

#### 5.5.5.5 Association Class

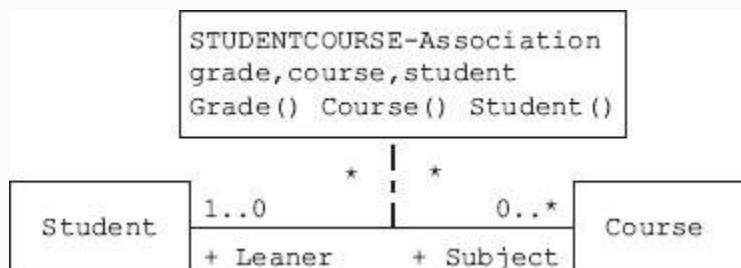
Association classes are required to be created, when there is a set of data that does not belong strictly to any one of the participating classes. For example, if there are two classes like student and course, then to which class do grade and attendance belong? There are several different grades gained by student and different attendance scores one for each course. Where do we store this kind of data? Where to place the GRADE? If we place it in student, the student will get the same grade for all courses. On the other hand, if we place the grade in course file, then all students taking the same course will get the same grade.





**Figure 5.13a** A student can undertake 0 to many courses

The solution is creating an Association class. An Association class will have its own attributes and methods, pointers and references to both of the participating classes.



**Figure 5.13b** An association class

#### 5.5.5.6 Composition

A class can contain other classes. For example, Student class can contain Date class. This can be viewed as whole–part

relation. The life cycle of part is dependent on the life cycle of whole. It also means that when the whole is deleted, the part also gets deleted. To decide about composition, the questions that are to be asked are:

- Is this class part of some other class?
- Is this class destroyed when some other class gets destroyed?

#### 5.5.5.7 Aggregation

Here, the part is not destroyed when the whole is destroyed. This is equivalent of a "has" type of relation. For example, a computer has a microprocessor. When the computer is destroyed, the microprocessor may not be destroyed. Another example of aggregation is: an automobile has an engine, wheels, etc. The critical question to be asked is: Is this class part of another class **and** is it independent of the other class?

#### 5.5.5.8 Generalization/Specialization

The generalization and specialization stems from inheritance relationship. Base class can be considered as a generalized class whereas a class extended from the base class is a specialized derived class. This association is

commonly referred to as inheritance because the derived classes inherit the functionality of their base classes to provide specialized behaviour. Inheritance provides the mechanism for new classes to be formed from the attributes and behaviour of existing classes. This is a "is" kind of relationship. The key question to be asked to identify inheritance relationships are: Is this class a specialization of a more general class?

#### **5.5.5.9 Interface**

An interface provides a common specification for behaviour, but, unlike an inherited class, an interface cannot be created or instantiated. An interface simply specifies common behaviour that other classes inherit. This means that interface has only method names and no implementations. Implementation is left to inherited classes from the interface which implement these methods differently. This means that unrelated classes can provide independent implementations. This is termed as run-time polymorphism.

## 5.6 OOAD: A Case Study

The first activity to be performed is requirement gathering. A series of meetings with users gives us a narrative of the problem involved. We are here to solve by designing an automated grievance redressal mechanism for a college. We would call this system as **GRC System (GRCSYS)**. Once the narrative is available, we do a grammatical parsing to identify nouns, verbs, etc. We have shown nouns and verbs in the narrative below. Nouns, we have shown in bold and underlined while verbs are in italics and bold.

Grievance Redressal Committee (GRC) of a university, headed by Chairman Grievance Redressal Committee (CGRC) *inquires* into **grievance** of students regarding girl-related matters, ragging, academic matters, and campus infrastructure-related aspects, and hostel and mess. A **Student** can *log on to* GRC System (GRCSYS) web site available both on **College Intranet** and **Internet Website (GRCSITE)** and *report* the **grievance**, indicating **grievance details**

(GRCDET) such as roll Number, class, course, type of grievance (TGRC), and date of occurrence (DOC) of grievance. They are *allocated* a token number (TNO) of the grievance. For *operating* web account, each student is provided with a password.

When a complaint is *received* on the web, CGRC *calls for* GRC. GRC in turn *looks into* the grievance and in *consultation* with concerned faculty and or Department *organizes* events such as *organizing extra coaching or special training, taking action on defaulting students and other necessary actions*.

Action initiated report (AIR) and action taken report (ATR) are *initiated* and *informed* to Student via web site. A suitable interface is provided for *displaying* on the web site to Students and Parents *to get* the reports of AIR and ATR, called AirReport and AtrReport with in a minimum time prescribed (TimePeriod).

Develop object-oriented analysis model for the above narration.



## Step 1: Identifying Object-oriented Candidate Classes

Parsing of narrative presented above, we can identify the following Candidate classes:

Grievance Redressal Committee (GRC)	System (GRCSYS)
CollegeIntranet	GRCSITE
Chairman Grievance Redressal Committee ( <u>CGRC</u> )	Student
Grievance	Class
Course	DOC
GRCDET	password.
TGRC	AIR
Token Number (TNO)	time period (PERIOD)
Faculty	Department
Event	AIR

ATR	
-----	--

Once the list is made, analyse the candidate classes. Look for collaborating classes. How does one class relate to another class? If it is collaborating then retain the class else discard it. Coad and Yourdon in their study suggested the following criteria to convert candidate classes into design classes:

1. Does the class have retained information?
2. Is the service provided by the class needed by other classes?
3. Does the class have multiple attributes?
4. Does the class have common attributes that are useful to others?
5. Does the class have common operations that are useful to others?
6. Does the class come under essential category and is required by other classes?

**Step 2: Let Us Apply the Above Criteria to Our Candidate Classes**

System (GRCSYS)	: Accept: Rules 1 to 6 apply
CollegeIntranet	: Accept: Rules 1 to 6 apply
Grievance Redressal Committee (GRC)	: Reject: Rules 1 and 2 not applicable
Chairman Grievance Redressal Committee ( <u>CGRC</u> )	: Reject: Rules 1 and 2 not applicable
GRCSITE	: Accept: Rules 1 to 6 apply
Student	: Accept: Rules 1 to 6 apply
Grievance	: Accept: Rules 1 to 6 apply
Class	: Reject: Rules 1 and 2 not applicable
Course	: Reject: Rules 1 and 2 not applicable
DOC	: Reject: Rule 3. Single attribute



GRCDET	: Accept: Rules 1 to 6 apply
password	: Reject: Rule 3. Single attribute
TGRC	: Reject: Rule 3. Single attribute
AIR	: Accept: Rules 1 to 6 apply
ATR	: Accept: Rules 1 to 6 apply
Token Number (TNO)	: Reject: Rule 3. Single attribute
time period (PERIOD)	: Reject: Rule 3. Single attribute
Faculty	: Reject: Rules 1 and 2 not applicable
Department	: Reject: Rules 1 and 2 not applicable
Event	: Accept: Rules 1 to 6 apply

## Step 3: Make a List of Design Classes

System (GRCSYS)	CollegeIntranet
GRCSITE	Student
Grievance	GRCDET
AIR	AIR
Event	

Let us consider **System (GRCSYS)** to work out further OOAD processes. We leave specifying operations and attributes as an exercise to readers.

## Step 4: Specify Attributes

## For specifying the class attributes

1. Look for descriptive attributes.
2. What characteristics distinguish this class from other classes?
3. Names are used to uniquely identify the objects. What characteristics uniquely identify this object?
4. How is a particular object linked to other objects?

---

```
roll Number, class, course, type of  
grievance (TGRC), date of occurrence  
(DOC) of grievance. token number (TNO),  
password, AirReport and AtrReport
```

---

## From the above list identify the attributes that belong to System (GRCSYS)

---

```
roll Number, class, type of grievance  
(TGRC), token number (TNO), password,  
AirReport and AtrReport
```

---

## Step 5: Specify Operations

For specifying the class operations, identify the operations each class performs. This list stems from the responsibility of the class. Operations specify the behaviour of the class. The operations can be classified as

- Modifier operations – These operations modify the attributes.
- Checks status of operations through checking a few attributes value.
- Transformation operation runs an algorithm or formula.
- Operation to check occurrence of an event.

ToInquire()	ToLogOn()	Toreport()
ToAllocate()	ToOperate()	ToProvide()
ToCallGRC()	ToOrganize()	ToConsult()
ToInitiate()	ToInform()	ToDisplay()

**From the above list we can identify the operations of System (GRCSYS):**

ToInquire()	ToAllocate()
-------------	--------------

ToProvide()	ToCallGRC()
ToCallGRC()	ToOrganize()
ToConsult()	ToInitiate()
ToInform()	ToDisplay()

Now we can complete the class diagram for **System (GRCSYS) class**

<b>System(GRCSYS):</b> roll Number class, type of grievance(TGRC), token number (TNO), pass word, AirReport AtrReport
ToInquire() ToOrganize() ToAllocate() ToConsult() ToProvide() ToCallGRC() Toinform() ToDisplay().

## 5.7 Design for Reuse

Designing software project using OOAD principles is a complex process and design OOAD project with reusability features is

even more difficult. But because of the enhancement of productivity, we need to use reusability features. The reusability features concentrates on two major aspects:

- **Reusability of code:** This is a built-in language feature and uses extendibility features of object-oriented language features such as composition, aggregation, inheritance, interface, etc.
- **Reusability of design:** This aspect concentrates on building software domain architecture as a framework for reuse activities. It is much more than mere code reuse. It is a domain engineering practice.

The idea behind reusability plank is not to try to derive and design from the first principles. Rather use the tried and tested classes that have gone through refinements. These classes form a “design pattern.” So what is a design pattern? First of all, the problem should have occurred repeatedly and there should be a recognizable pattern, so that under similar situations, the design pattern could be made use of for elegant solutions. In other words, design pattern allows us to choose from design alternatives and make the system reusable. Design patterns have the following characteristics.

- **Pattern Name:** Identifies the design pattern for reuse. Pattern name describes a design problem, its solutions, its consequences a word or two.
- **Problem:** Describes the problem and when to apply a particular design pattern. For example, problem describes specific design problems of representing algorithm as object, etc. The problem may include conditions that must be met before we apply the design pattern.
- **Solution:** Elements of design, their relationships, responsibilities and collaborations. Solution cannot be an exact solution since design pattern is like a template which has to be fitted for a particular problem. The design pattern provides an abstract description of a design problem and arrangement of classes and objects to solve the problem.
- **Consequences:** There are consequences of applying a design pattern to a problem. Listing of these consequences help a designer to choose from the alternatives.

## How can design pattern help a designer?

- To identify less obvious objects such as a process or an algorithm.
- To decide what should be an object – in other words, object granularity.
- To specify object interfaces.
- To specify object implementations.

## Principles of Reusable Object-oriented Design

- Inheritance extends a class functionality and one can get a new class merely by extending an existing class.

- Due to polymorphic property, all derived classes share the same interface. The users of interface are governed only by interface and are generally unaware of implementation by derived class. This is a great feature of reuse. We can program to an interface rather than an implementation.
- Inheritance by extending functionality can be seen as breaking encapsulation whereas composition has well-defined interfaces for the objects being composed and hence implementation is hidden because of interface. Hence composition is a better reusability feature than inheritance.
- Generic programming using templates is a third method for reusability feature, in addition to inheritance and composition. Parameterized templates allow us to define a type without specifying all other types. The type can then be passed at run-time as an argument. For example, we can pass "integer" as a type to list parameterized type.

## 5.8 Comparison of SASD and OOAD Methodologies

- Step-by-step stagewise development and documentation.
- Both techniques use graphical design and graphical tools such as CASE and UML to analyse and model the requirements.

### Differences

- SASD is process-oriented.
- OOAD is data-oriented.
- OOAD encapsulates Data and Process.



- Heavy reliance of OOAD on reusable components.
- In SASD, the user requirements are collected and specifications are drawn based on requirements, and users are then asked to sign off on the specifications.
- In OOAD, the requirements and specifications are matched repeatedly over the entire development period and users are involved at each stage.

## 5.9 Summary

1. There are four phases in project management, namely: inception, elaboration, construction and delivery.
2. Waterfall model of software development life cycle closely follows structural analysis and design methodology and is widely used in industry.
3. Structured analysis and design (SASD) methodology is used to convert business model into specifications that can be used to develop computer-based programs.
4. SASD technique attempts to solve the complex problem by dividing large, complex problems into smaller, more easily handled ones. The technique can be called the divide and conquer method. The approach adopted can also be called the top-down approach.
5. SASD comprises data modelling and functional modelling. Data flow and control are depicted using DFDs.
6. The main elements of SASD are: (1) statement of purpose, (2) context diagram, (3) event list, (4) DFDs, (5) data dictionary, (6) entity relation diagrams, etc.
7. Context diagram of SASD is a set of inputs and outputs and a transformation system.
8. The purpose of event list is to prepare a list of external events/activities to which system under design should respond.

9. Data flow diagrams (DFDs), also called data flow graphs, are used in the analysis phase. A DFD views function as data flows through the system. DFD highlights the data transformation from input stage to output stage through various processes.
10. A data dictionary or **database dictionary** is a file that defines the database. Data dictionary only maintains information to manage data but holds no actual data. We can call it meta data about data but not actual data.
11. Entity relationship diagram (ERD) is a graphical representation of the data layout of a system at a high level of abstraction. It defines data elements and their inter-relationships in the system.
12. Process specifications show process details, which are not shown in a DFD. Input, output and algorithm of a module in the DFD are included in process specifications.
13. Functional Decomposition: In order to be effective, we need to specify constraints and operations so that original specifications and requirements are not compromised because of functional decomposition.
14. Object-oriented classes support the object-oriented principles of abstraction, encapsulation, polymorphism and reusability. Classes are specifications for objects.
15. Classes are derived from the use cases or problem statement or narrative; classes provide an abstraction of the requirements and provide an internal view of the application.
16. The stages in OOAD are: (1) identifying classes, (2) identifying attributes, (3) specifying operations and (4) work out associations and relationships.
17. A relationship is an association between classes. Relations between classes occur when a class has some service to offer to other classes or uses services offered by other classes.

18. Association classes are required to be created, when there is a set of data that does not belong strictly to any one of the participating classes.
19. In composition, a class can contain other classes. For example, Student class can contain Date class. This can be viewed as a whole–part relation. It also means that when the whole is deleted, the part also gets deleted.
20. In aggregation, the part is not destroyed when the whole is destroyed. This is equivalent of a “has” type of relation.
21. The generalization and specialization stems from inheritance relationship. Base class can be considered as a generalized class whereas a class extended from the base class is a specialized derived class.
22. An interface provides a common specification for behaviour, but, unlike an inherited class, an interface cannot be created or instantiated. An interface simply specifies common behaviour that other classes inherit.
23. Reusability revolves about reusability of code and reusability afforded through the use of design pattern.
24. A design pattern is a pattern formed when a problem occurs repeatedly and there is a recognizable pattern, so that under similar situations, the design pattern could be made use of for elegant solutions. In other words, design pattern allows us to choose from design alternatives and make the system reusable.
25. Inheritance, composition and parameterized templates are the tools provided by objectoriented languages to ensure reusability features.

## Exercise Questions

### Objective Questions

1. Specifications are deliverables for

1. feasibility study
2. analysis phase
3. design phase
4. problem definition.

2. Which of the following statements are NOT true in case of the waterfall model?

1. Resources can be committed incrementally and not all at once.
2. Stagewise documentation is available.
3. It follows SASD design methodology.
4. It is an incremental model.

3. Which of the following statements are true?

1. OOAD is process oriented
2. SASD is process oriented
3. OOAD encapsulates data and process
4. SASD models requirements

1. i
2. i and ii
3. i, ii and iii
4. ii, iii and iv

4. Which of the following statements are true with respect to SASD?

1. Developed in the late 1970s by DeMarco, Yourdon and Constantine
2. SASD is process oriented
3. SASD models requirements
4. SASD takes care of both functional and non-functional requirements

1. i
2. i and ii
3. i, ii and iii
4. ii, iii and iv

5. Scope of the project is defined by

1. Statement of purpose
2. Context diagram
3. Narrative
4. Event list

6. Which of the following is false with respect to context diagram?

1. Context diagram defines the purpose of the project.
2. Interaction of external elements with system under development.
3. Shows inputs, outputs and transformation functions.
4. Context diagram and data flow diagram are one and the same.

1. i and iv
2. i and iii
3. i, ii and iii
4. None

7. Which of the following are true?

1. DFDs are functional decomposition of the main process.
2. DFDs are graphical representations of the decomposition process.
3. DFDs describe the data required to meet functional specifications.
4. DFDs depict data flow but do not show control flow.

1. i
2. i and ii
3. i, ii and iv
4. ii, iii and iv

8. Which of the following are true?

1. DFDs are useful in the analysis phase.
2. DFDs are useful in the design phase.
3. DFDs describe functions as data flows.
4. DFDs describe data transformation.

1. i
2. i and ii
3. i, ii and iv
4. i, ii, iii and iv



9. Which of the following are false regarding data dictionary?

1. Data dictionary is a list of files and databases.
2. Data dictionary holds actual data.
3. Defining data dictionary is optional.
4. Data dictionary describes data transformation.

1. i
2. i, ii and iii

- 3. ii, iii and iv
- 4. i, ii, iii and iv

10. Which of the following statements are true with respect to ERDs?

- 1. ERD is a graphical representation.
- 2. ERD is a data layout and their inter-relations between data elements.
- 3. ERD symbol.  represents relationship.
- 4.  symbol represents data element.

- 1. i
- 2. i, ii and iii
- 3. ii, iii and iv
- 4. i, ii, iii and iv

11. Which of the following statements are true regarding OOAD?

- 1. Classes are specifications for objects.
- 2. CRC cards can be used to decide the classes in OOAD.
- 3. A relationship is an association between classes.
- 4. Association can both not have multiplicity of roles

- 1. i
- 2. i, ii and iii
- 3. ii, iii and iv
- 4. i, ii, iii and iv

12. Which of the following statements are true with respect to OO language?

- 1. Association class is created when there is data that does not belong strictly to a single class.
- 2. Composition depicts a whole-part relationship.
- 3. A composition depicts a "has" type of relationship.
- 4. In aggregation, part is destroyed when the whole is destroyed.

- 1. i and ii
- 2. i, ii and iii
- 3. iii and iv
- 4. i, ii, iii and iv

13. Which of the statements are false with respect to inheritance relationship?

1. Inheritance depicts a "has" type of relationship.
2. Inheritance depicts an "is" type of relationship.
3. Generalization refers to derived class in inheritance relationship.
4. Specialization refers to derived class in inheritance relationship.

1. i and iv
2. i, ii and iii
3. ii, iii and iv
4. i, ii, iii and iv

14. Which of the following statements are true with respect to reusability in OOAD?

1. Reusability uses design patterns.
2. Reusability means both reusability of code and reusability of design.
3. Inheritance is better for reusability feature than composition.
4. Interface is a feature that helps greatly in achieving reusability.

1. i and iv
2. i, ii and iii
3. ii, iii and iv
4. i, ii, iii and iv

15. Which of the following features aid reusability features of OO language?

1. Parameterized templates
2. Composition
3. Inheritance
4. Interface

1. i, ii and iii
2. i, ii and iv
3. i, ii, iii and iv
4. ii, iii and iv

#### **Short-answer Questions**

16. What are the stages in project management?
17. What are the main elements of SASD?
18. Explain the purpose of context diagram in SASD.  
Explain the notations used in context diagram?
19. What is a data dictionary and why is it required?
20. What is statement of purpose? Is it a clear and concise statement highlighting the purpose of the system?

21. What are the elements of SASD?
22. What are the elements of data dictionary?
23. What is ERD?
24. What are the symbols used in ERDs?
25. Why are specifying process and constraints important in SASD?
26. Explain Composition, Aggregation and Association class.
27. Explain the terms generalization, specialization, interface and design pattern.

#### **Long-answer Questions**

28. Explain the waterfall model of the software development life cycle. What are its advantages and disadvantages?
29. Explain functional decomposition in SASD. What is the role of DFD in functional decomposition and analysis?
30. Explain the role and usefulness of ERDs in SASD.
31. What are the advantages and disadvantages of SASD?
32. When do you use SASD?
33. When do you use OOAD?
34. Distinguish between SASD and OOAD techniques.
35. What is an association in OOAD? Explain with examples.
36. Distinguish between composition and aggregation.
37. Distinguish between inheritance and interface. What are generalization and specialization? How are they related to inheritance?
38. What are design patterns? Explain their contribution to reusability features of a modern object-oriented language like Java or C++.
39. What are the advantages and disadvantages of using SASD methodology? Explain the areas SASD methodology is efficient.

#### **Assignment Questions**



40. Write a narrative for students admission process in a professional college and do grammatical parsing to identify classes and their associations.
41. Write a narrative for college academic administration of a professional college. Include areas like departments offering courses, students registering for a course, professor handling courses, students' performance in the form of grades, etc. Draw class diagram, depicting associations and relationships.
42. Write a narrative for book ordering system of a library. Identify the objects using grammatical parsing techniques.
43. Carry out object-oriented analysis and design for online purchase of items from an Internet-based E-Shop.

### **Solutions to Objective Questions**

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



# 6

## C++ Fundamentals and Basic Programming

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to understand and put to use*

- Development of C++ and OOPS as applied to C++ language.
- C++ programming environment and programming structure.
- Curricular programs in C++ using control loop, arrays, structure and classes. These programs are designed to give you a taste of things to come and also to make you productive from day one.
- Console IO operations and unformatted functions used in input and output operations.

## 6.1 Introduction

In order to solve a problem, you need know the “method/procedure” or have the “know-how.” In computer parlance, this can be called an algorithm. An algorithm or program is a sequence of steps to be followed, which, when followed, leads to a desired output. The sequence of steps can be called a procedure. A group of procedures can be termed as a program.

For solving a complex problem, we need a programming environment that includes wellintegrated and cohesive programming elements, constructs and data structures. The structured programming paradigm with C as implementing language, in which the programmer breaks down the task to be accomplished into smaller tasks and specifies step-by-step procedures or algorithms to achieve the task at hand, has been very successful and popular.

Low productivity of programmers due to factors such as changing user needs, complexity of projects and non-availability of extensibility and reuse features is the

main factor for projects not being completed on time. Object-oriented programming (OOP) promises extensive reusability features through inheritance, library functions in the form of standard template library (STL), etc. C++ is one of the most powerful OOP languages that supports both structured as well as OOP paradigms.

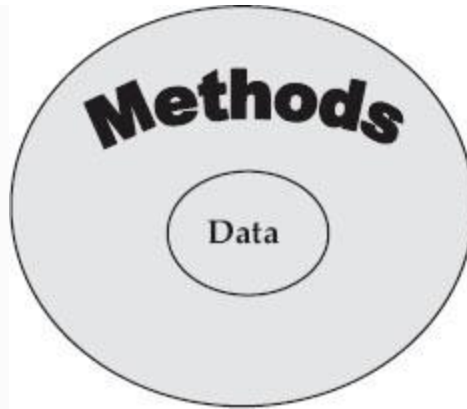
## 6.2 History and Development of Object-oriented Languages Like C++

With the advent of UNIX and communication hardware and software from 1969 onwards, the complexity of both hardware and software kept on increasing and software project development could not keep pace. As a result, the quality of programs suffered. The developers at Bell Laboratories and the Department of Defense, US, started to look at alternative paradigms wherein reusability and extensibility were strong features. Further, the specifications laid insisted on data primacy rather than process primacy.

C++ was developed by Bjarne Stroustrup in 1979 at Bell Laboratories. C++ is a sequel to the C language, with additional enhancement features, including virtual functions, function name and operator overloading, references, free space memory, and several features on reusability and extendibility through inheritance, templates, etc. Exception handling and a welldeveloped STL are two of the advanced features available in C++.

### 6.3 Object-oriented Language Features

OOP style can be of two types, i.e. object-based programming and object-oriented programming. In OOP paradigm, object is primacy. Object-oriented programming, such as C++, uses objects and interactions amongst them through invoking member functions. Figure 6.1 shows object holding data and member functions. The features include data hiding, data abstraction, encapsulation, inheritance and polymorphism.



**Figure 6.1** Data and methods in OOP paradigm

## **Salient Features of the OOP Paradigm**

- Data encapsulation
- Data hiding
- Operator overloading
- Initialization and automatic clearing of objects after use
- Dynamic binding
- Extensive and well-defined standard template library

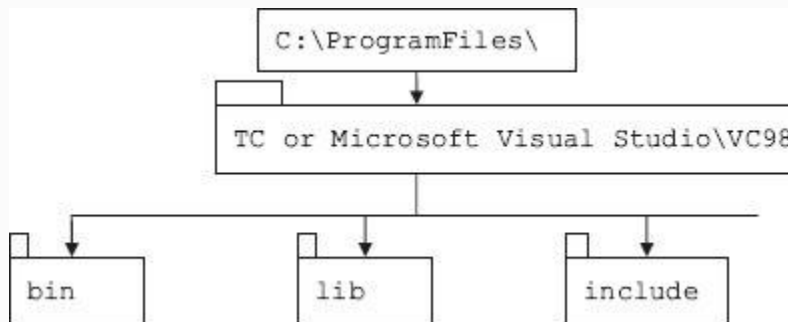
### **6.4 Welcome to C++ Language**

We want to make you reasonably comfortable with C++ language. The idea is that you should be productive from day 1. Get ready for an exciting tour of C++ in the introductory chapter itself. We will introduce C++ through simple programs

after highlighting a few essential features you should know before you write your first C++ program.

### *6.4.1 Setting Path*

After installation of Turbo C++ or Micro Soft Visual C++ or GNU C++ Compiler for Linux platforms, the relevant files will normally get loaded into `C:\Program Files\TC` or `C:\Program Files\Microsoft Visual Studio\VC98` unless otherwise chosen by you. After downloading, the directory will look as in [Figure 6.2](#).



**Figure 6.2** Directory structure after installing TC or VC++



Let us say that we want to use folder called "oops++" in D directory and within the folder we have created a separate folder called "c++chap1". In this folder, we will create all our source code files and also store all object files and other related project files created by VC++.

Therefore, the path for storing all class files and source programs developed by us would be

---

- D:\oops++\
- 

We also need to link up bin and lib directories of TC\VC++. Their paths would be

---

- C:\Program Files\ Microsoft Visual Studio\VC98 \bin // contains all company //supplied programs
- C:\Program Files\ Microsoft Visual Studio\VC98\include; // win32 & header files
- C:\Program Files\ Microsoft Visual Studio\VC98\lib // libraries

Or

- C:\Program Files\ TC\bin

- `C:\Program Files\ TC\include;`  
`C:\Program Files\ TC\lib`
- 

## **Example 6.1: How to Set Path for Windows Platforms**

### **Permanent setting of path variable:**

---

- Go to settings->control panel->system -> advanced-> set environment variable.
  - Select path->edit : copy & paste path -> ok
- 

## **Example 6.2: How to Set Path for Linux Platforms**

Let us suppose that TC or VC++ has been installed in directory `/usr/local/`

## **Linux :**

**For C shell (csh), edit the startup file (~/.cshrc) :**

```
set path=(/usr/local/TC/bin )
```

**For bash, edit the startup file (~/.bashrc) :**

```
PATH=/usr/local/TC/bin:  
export PATH
```

**For ksh, the startup file is named by the environment variable, ENV.**

```
To set the path:  
PATH=/usr/local/TC/bin:  
export PATH
```

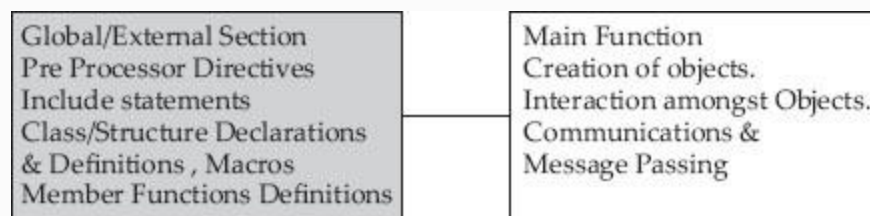
**For sh, edit the profile file (~/.profile) :**

```
PATH=/usr/local/TC/bin:  
export PATH
```

### ***6.4.2 C++ Program Structure***

Any C++ program contains sections shown in Figure 6.3. Global section, also called

external section, contains subsections which are available and visible to all parts and functions and sections of the program. Preprocessor directives are instructions to compilers to perform certain tasks before actual compilation of source code takes place.



**Figure 6.3** Structure of a C++ program

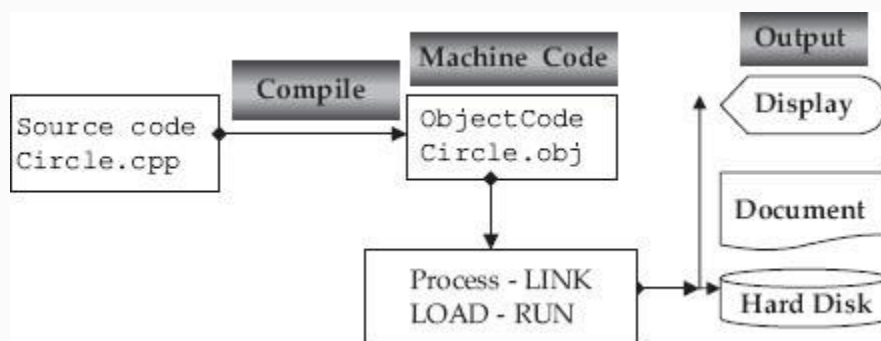
**Global/External Section:** For example, `#include<stdio.h>` statement gets the file from include section of the library created by compiler and attaches it with your code, and then the source code is compiled. Declarations of user-defined data types such as class and structures are included. Member functions and macrofunctions are declared and defined in

this section. They are available to the entire program.

Function `main()` is mainly used to create required objects and variables and as interface with the user. It is used to obtain input data and displaying the final results. Objects created interact with other objects by invoking functions and passing arguments and achieve the desired result.

### 6.4.3 C++ Development Environment

Before we execute our program there is a need to understand the C++ development environment. Refer to [Figure 6.4](#).



**Figure 6.4** C++ development environment

## Step 1: Edit

Edit the source program using any of the editors notepad. For example, our program will have a main class "HelloWorld".

Hence, our Java program also to be named as HelloWorld.cpp Move to directory d:\oopsc++\c++chap1 and create the required source file. Use any text editor like notepad, vi editor, etc., and enter your code and save it as HelloWorld.cpp.

```
d:\oopsc++\c++chap1> edit  
HelloWorld.cpp
```

---

```
1. /* HelloWorld.cpp. This is our  
first program to introduce you  
2. to some basic io statements like  
cin and cout & multi line comment*/  
3. #include<iostream> //iostream, a  
library file contains cin , cout  
programs  
4. using namespace std;  
5. void main( ) // start of main  
programme  
6. {  
7. char name[20]; // name is a data  
type of char of c++ length 20  
8. cout<<"\n Please enter your name :
```

```

"; //cout is an object of iostream
9. cin>>name; // cin is an object of
iostream cout<<"\n Hello "<<name<<endl;
//\n means goto new line
10. cout<<"\n Welcome to Objective
Oriented Programming in C++"<<endl; //
endl means goto new line
11. cout<<"\n Best Wishes from
authors. Have a good day!"<<endl;
12. }// end of main

```

---

**Lines No. 1 and 2:** C++ provides two types of comment statements. Comments are included to enhance the readability of the program. C++ allows both single-line comment statements and also multiline comment statements.

- **Single-line comments:** These comment lines start with double slash : //
- Ex: //Circle.cpp, a C++ cprogram to print a line of text
- **Multiline comments:** These comment lines are used when comments extend beyond a single line. They start with /\* and end with \*/
- Ex: /\* Inheritance.cpp :The programme introduces concepts of

**Inheritance and other related reusability features of C++ \*/**

## Global Declarations and Definitions.

All statements included before `main()` function. All include sections, define statements, function prototype declarations, and structure definitions are global declarations. Hence they are available to all functions also.

**L** `include<.....>` statements are preprocessor  
**i** directives. `iostream` is included, so that  
**n** all functionality provided by `iostream`, i.e. input  
**e** and output stream like `cin` and `cout` are  
**N** available to us. If you are using Turbo C++  
**o** compiler you will use  
• `#include<iostream.h>`.

**3**  
**:**

**L** Using name space `std;` is a statement used to  
**i** resolve conflicts arising due to the same names  
**n** being used in different programs. Here we are  
**e** instructing the compiler to use standard library  
**N** supplied by compiler. For Turbo C++ users,  
**o** using name space `std;` statement is not  
• required.

**4**  
**:**



**L** void main() is start of main function and  
**i** program. Void is a data type that a function  
**n** returns after executing the function. Observe  
**e** that start of main is indicated by an opening  
**N** brace bracket and closing brace at line no 12.  
**o**  
**.**  
**5**  
**:**

**L** Name has been declared as data type char of  
**i** length 20. We can hold name in this field.  
**n**  
**e**  
**N**  
**o**  
**.**  
**7**  
**:**

**L** Cin and cout are objects of iostream used to  
**i** input and output data using iostream.. \n  
**n** means printing starts in new line. What has  
**e** been written with in " " is printed as it is on to  
**s** output console. >> & << are extraction  
**N** operators in iostream. They are used to stream  
**o** data in and out with cin and cout, respectively .  
**.** <<endl is a command directing compiler to go  
**8** to new line.  
**—**  
**1**

<b>1</b> :	
---------------	--

## Step 2: Compile

We need to store all our compiled code in `d:\oopsc++\c++chap1`. Therefore, from the directory `d:\oopsc++\c++chap1`.

Compile the source file

Compile it using `# gcc`

`HelloWorld.cpp` Output using `# a.out`  
for Linux based system

---

```
Run ----- Compile -----  
Execute for Turbo C
```

---

If you are using `vc++` platform, the job of compilation and link load and run are fairly simple. On the task bar choose: compile, build and run buttons.

If you are using Linux environment:

1. Switch on the computer.
2. Select the Red hat Linux environment.
3. Right click on the desktop. Select 'New Terminal'.

4. After getting the \$ symbol, type '**vi filename.cpp**' and press Enter.
  5. **Press Esc+I** to enter into Insert mode and then type your program there.  
The other modes are Append and Command modes.
  6. After completion of entering program, press (**Esc + Shift + :**). This is to save your program in the editor.
  7. Then the cursor moves to the end of the page.  
**Type 'wq' and press Enter.**  
(wq=write and quit)
  8. On \$ prompt type, c++ **filename.cpp** and press Enter.
  9. If there are any errors, go back to your program and correct them.  
Save and compile the program again after corrections.
  10. If there are no errors, run the program by typing  
**./a.out and press Enter.**
  11. To come out of the terminal, at the dollar prompt, **type 'exit'** and press Enter.
- /\* Out put**  
Please enter your name: Ramesh  
Hello Ramesh  
Welcome to Objective Oriented Programming in C++  
Best Wishes from authors. Have a good day! \*/

## 6.5 Further Sample Programs of C++

### 6.5.1 Execution of a Program in a Loop

The second problem is finding surface area of a. We would like to compute the surface area of a football that is spherical in shape, given its radius, given by the formula  $4 \pi r^2$

$PI * r * r$ . In this section, we will use a while loop to run the program. The code for the above problem is given below:

### **Example 6.3: SurfaceArea.cpp. to Compute Surface Area of a Football**

#### **//Example 6.3 SurfaceArea.cpp Program to calculate surface area of a ball**

---

```
1. #include<iostream>
2. using namespace std;
3. /* Declare function prototype that
   will compute the surface area given the
   radius of float (real number) data type
   and returns area again of float data
   type
4. float FindArea(float radius , const
double PI);
5. void main( )
6. { // You will need float (real
numbers) data types for holding radius
and area
7. float radius, area;
8. const double PI=3.1415926; // PI
```

```

value is constant and does not change
    9. // Obtain radius from the user.
User can enter 0 to stop the programme*/
    10. cout<<"\n Enter radius of the
ball. To stop enter 0 for the radius
\n";
    11. cout<<"\n Radius = ? ";
    12. cin>>radius; // user enters value
for radius
    13. /* We would like to compute
surface areas of different balls. So
user enters various radii. User can stop
by entering 0 for the radius. We will
use while statement.*/
    14. while (radius != 0) // i.e. till
user does not enter 0
    15. {
    16. if (radius < 0) area = 0; // if
radius is negative area =0
    17. else area = FindArea(radius,PI);
// calling out function. Radius & PI are
arguments // We store the result
returned by FindArea() at area.
    18. // print out put
    19. cout<<"\n Surface area of football
:"<<area<<endl; //
    20. /* what is in quotes appears as it
is in the output. \n means leave a line
after print
    21. We are inside while statement. We
have just completed printing of area.
    22. What should we do next? Ask the

```

```

user for next problem i.e. next radius.
Ask it.*//
    23. cout<<"\n Enter radius of the
ball. To stop enter 0 for the radius
\n";
    24. cout<<"\n Radius = ? ";
    25. cin>>radius;
    26. }// end of while
    27. }// end of main
    28. // function definition. Here we
will tell what FindArea function does
    29. float FindArea (float radius,const
double PI)
    30. { float answer; // we will store
the area
    31. answer = 4* PI * radius * radius;
    32. return answer;
    33. } // end of function definition
/* OUTPUT : Enter radius of the
ball. To stop enter 0 for the radius
Radius = ? 2.0
Surface area of football :50.2655
Enter radius of the ball. To stop
enter 0 for the radius
Radius = ? 3.0
Surface area of football :113.097
Enter radius of the ball. To stop
enter 0 for the radius
Radius = ? 0
Press any key to continue*/

```

---



---

**L** Function Prototype like float **FindArea**  
**i** **(float radius, const double PI);** tells  
**n** the compiler that function definition is after  
**e** **main( )** but its usage will be in the **main( )**.  
**N** It is like advance information to the compiler to  
**o** accept usage prior to definition. We could also  
define the above function as:

**·** float FindArea(float radius ,  
**4** const double PI)  
**:** {return 4\*PI\*radius\*radius;}

We will be using this second and short form  
for convenience throughout the book.

Function definition has been provided at Line  
No 28.

**L** We have declared PI as constant and of data type  
**i** double. Double means it is double precision data  
**n** type.

**e**

**N**

**o**

**·**

**8**

**:**

### 6.5.2 Arrays Implementation

In this example, we will show the use of arrays. Array is a data structure comprising of contiguous memory location. It will store the same data type in all locations. In Example 6.5.1, we have taken in the radius and immediately declared the result and continued the process till user entered 0 to exit. In this example, we will take all the radii and heights of several sizes of cones (ice cream?) together and store them in memory location, compute corresponding volumes of these cones and store them as well and declare the results in the end. Array comes to our rescue here. Array is used to hold the same data type in contiguous memory locations. We will define three arrays named radius, height, and volume, each with a dimension of 10 elements as shown in Figure 6.5.



float radius[10];										
	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0
radius	0	1	2	3	4	5	6	7	8	9
float height[10];										
	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0
heights	0	1	2	3	4	5	6	7	8	9
float volume[10];										
	?	?	?	?	?	?	?	?	?	?
volume	0	1	2	3	4	5	6	7	8	9

**Figure 6.5** Arrays for radius heights and volume

In C++ language, it is customary to number the cells of the array starting with 0. Accordingly, `radius[0]` holds a value 2.0 and `radius[9]` holds a value of 11.0. `height[0]` holds a value of 5.0 and `height[9]` holds a value of 14.0. The formula for volume of cone is  $4 * \text{PI} * \text{radius} * \text{radius}$ . We will store the values of cones in an array called volume. With this knowledge, let us attempt array implementation of cones.

## **Example 6.4: volume.cpp. to Compute Volumes of Ice Cream Cones**

`/* Example 6.5.2 volume.cpp. a program to compute volumes of cone using arrays. We will store all the radii and heights and compute the volume and store them in an array called volume. When user terminates the program by entering 0 for radius, we will declare all the results.*/`

---

```
1.  #include<iostream> // for using
    cin and cout from library.
2.  using namespace std; // to resolve
    naming conflicts. Use from standard
    library.
3.  // Declare Function Prototype
4.  float FindVolume(float radius,
    float height, const double PI);
5.  void main( )
6.  { int n, i =0; // i we will use as
    array index, n for keeping count
7.  const double PI=3.1415296;
```

```

8.  float radius[50]; // Array of
radius, numbering max of 50
9.  float height[50]; // Array of
heights, numbering max of 50
10. float volume[50]; // Array of
volume, numbering max of 50
11. cout<<"\n Enter radius & height of
a cone <To stop enter 0 for the radius>
"<<endl;
12. cout<<" Radius & Height : ? ";
13. cin>>radius[i]>>height[i]; //Store
at address of radius[i]& height[i]
14. while (radius[i] != 0)
15. { if (radius[i] < 0)
        volume[i] = 0;
    else
        volume[i]=
FindVolume(radius[i],height[i],PI); //
result in array volume
16. // get the next set of data. we
have to increment i prior to getting
17. // new set of data else old data
will be overwritten and hence lost.
18. i++;
19. cout<<"\n Enter radius & height of
cone <To stop enter 0 for the
radius>"<<endl;
20. cout<<" Radius & Height : ? ";
21. cin>>radius[i]>>height[i];
22. } // end of while
23. n = -- i; // This is because you
have increased the count for i for

```

```

        // radius = 0 case also.
We will use n in for loop.
    24. // display array elements
    25. cout<<"\n Volume of Ice cream
Cones\n";
    26. // You have n cones (i.e. 0 to n-1
as per C++ convention).
    27. //Therefore print i <=n using a
for loop
    28. for (i=0; i<= n; i++)
    29. cout<<"
radius["<<i<<"]="<<radius[i]<<"
height["<<i<<"]="<<height[i]<<" :
volume["<<i<<"]="<<volume[i]<<endl;
    30. }// end of main
    31. // function definition
    32. float FindVolume(float
radius,float height, const double PI )
    33. {return (4*PI * radius *
radius*height);}

/*Output :Enter radius & height of
a cone <To stop enter 0 for the radius>
Radius & Height : ? 2.0 3.0
Enter radius & height of cone <To
stop enter 0 for the radius>
Radius & Height : ? 4.0 5.0
Enter radius & height of cone <To
stop enter 0 for the radius>
Radius & Height : ? 0 0
Volume of Ice cream Cones
radius[0]=2 height[0]=3 :
volume[0]=150.793

```

```
radius[1]=4 height[1]=5 :
volume[1]=1005.29*/
```

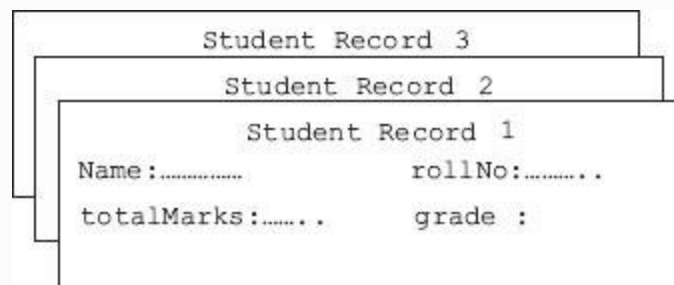
<b>Line No. 4:</b>	A function prototype takes radius, height and PI as input and returns volume.
<b>Line No. 8 – 10 :</b>	Declare arrays of maximum length $\leq 50$ .
<b>Line No. 13 :</b>	<code>cin&gt;&gt;radius[i]&gt;&gt;height[i] ;</code> inputs values for radius and height. Data values need to be separated by space while entering the data.

<b>Line No. 28:</b>	for (i=0; i<= n; i++) It is a header statement of for loop. Initial condition is i=0; Final condition for loop to terminate is i<= n; The loop will be executed in Steps of 1. Hence i++)
<b>Line No. 29:</b>	Displays Radius, height and volume. It is body of loop. Body of the loop is a single line.
<b>Line No. 32:</b>	Function definition to return volume.

### 6.5.3 Use of Structure to Implement Problem

While arrays store the same type of data in contiguous locations, structure can be used to store different types of data, bundled

together as an instance of structure. C++, in addition to object-oriented features, supports all features of C such as structures and unions. These find extensive use in data structures and other C-based software implemented in C++ environment. We can define structure in C++ language for records shown in Figure 6.6.



**Figure 6.6** Student record for three students

```
struct Student
{
    char name[20];
    int rollNo;
    float totalmarks;
    char grade;
};
```

```
students
// Stud typecasted to Student structure
typedef struct Student
```

---

Stud; We can refer to items inside the structure using

---

```
std[i].name, std[i].rollNo,
std[i].totalMarks std[i].grade
```

---

We have used typedef statement so that we can use Stud in lieu of lengthy *struct Student* every time we refer to Student structure. User enters name and total marks. If total marks are more than 80% declare as A grade, 60–79 % declare as B grade and below 60% declare as C grade.

**Example 6.5: StudStru.cpp. to Compute Grades of Students Using Structures**



---

```
1.  #include <iostream>
2.  using namespace std;
3.  // declare function prototypes
4.  char FindGrade(float totalMarks);
5.  // declare a structure
6.  struct Student
7.  {
8.  char name[20];
9.  int rollNo;
10. float totalMarks;
11. char grade;
12. };
13. // Stud typecasted to Student
structure
14. typedef struct Student Stud;
15. void main( )
16. { Stud std[3]; //maximum of 3
Students
17. int n, i=0; // i we will use as
structure index, n for keeping count
18. cout<<"\n Enter Roll Number <0 to
STOP> :";
19. cin>>std[i].rollNo;
20. while ( std[i].rollNo!=0) //
variable declared in std are referred
with dot(.)
21. { cout<<"\n Enter name & roll
number & total Marks< 200 to 600"<<endl;
22.
cin>>std[i].name>>std[i].totalMarks;
23. std[i].grade=
```

```

FindGrade(std[i].totalMarks);
    24. ++ i;
    25. cout<<"\n Enter Roll Number <0 to
STOP> :";
    26. cin>>std[i].rollNo;
    27. }// end of while
    28. n = -- i; // because you have
increased the count for i for name =STOP
case also.
    29. // display structure elements
    30. cout<<"\n Students Grades\n";
    31. for(i=0;i<= n; i++)
    32. { cout<<"\n name
:"<<std[i].name<<" "<<"Roll No
:"<<std[i].rollNo;
    33. cout<<" Total
Marks"<<std[i].totalMarks<<" "
    34. <<"Grade : "<<std[i].grade<<endl;
    35. }// end of for loop
    36. }// end of main
    37. // function definition
    38. char FindGrade(float totalMarks)
    39. {char grade;
    40. if ( totalMarks>=480)
    41. grade='A';
    42. else
    43. if ( totalMarks>=360)
    44. grade='B';
    45. else
    46. grade='C';
    47. return grade;
    48. } // end of function definition

```

```

/*Output : Enter Roll Number <0 to
STOP> :50595
Enter name & roll number & total
Marks<between 200 to 600
Ramesh 358
Enter Roll Number <0 to STOP>
:60695
Enter name & roll number & total
Marks<between 200 to 600
Gautam 540
Enter Roll Number <0 to STOP>
:75775
Enter name & roll number & total
Marks<between 200 to 600
Anand 540
Enter Roll Number <0 to STOP> :0
Students Grades
name :Ramesh Roll No :50595 Total
Marks358 Grade : C
name :Gautam Roll No :60695 Total
Marks540 Grade : A
name :Anand Roll No :75775 Total
Marks540 Grade : A */

```

**Lin  
es  
No.**

**Structure declaration. Observe structure  
declaration ends with };**

<b>6– 13:</b>	
<b>Line No. 14:</b>	Type casting of Stud to structure. Hence we can use Stud instead of struct Stud as in statement 16, which declares three instances of structure stud.
<b>Line No. 19:</b>	Variables of structure are always referred with <code>dot ( . )</code> operator.

### *6.5.4 Class Implementation*

In this section, we will introduce the concepts of class and objects. What is an object? An object is an entity that you can feel. For example, pen, student and football are all objects. Consider a class in which, let us say, there are 60 students who are interested in studying for C++ course. Then we can group all 60 students into a class, just like your college administration groups all its B.Tech computer science students into a class. In C++, we would declare a class. A class will have member data and member

functions. We call this as attributes and behaviour. Once a class is created, we will create an instance of class called object. Objects invoke functions and interact amongst themselves to achieve the desired result.

In Example 6.5, we will compute the total and grade of a student. A class called Student will have name and roll number, subject marks, total and average as private member data. As a policy of C++, all data is declared as private. When you declare a data as private, access is permitted only for members of the class. We will access these private data through public functions.

### **Example 6.6: stud1.cpp to Compute Total and Average and Display Result**

---

```
1. #include<iostream>
2. using namespace std;
3. //declare a class called Student
```

```

4. class Student
5. {
6. private:
7.     int rollNo;
8.     char name[30];
9.     double marks[5]; // set of five
subject marks
10.    double tot;
11.    double avg;
12. public:
13.    void GetData(int n);
14.    void ComputeAvg(int n);
15.    void DispData(int n);
16.};
17.// Define member functions
18.void Student::GetData(int n)
19.{ cout<<"\n Enter Students Details
"<<endl;
20.    cout<<" Enter Roll No <space>
name : ";
21.    cin >>rollNo>>name;
22.cout<<" Enter marks in "<<n<<"
subjects separated by spaces :";
23.for ( int i=0;i<n;i++)
24.cin>>marks[i];
25.}
26.void Student::DispData(int n)
27.{ cout<<rollNo<<" "<<name<<" ";
28.for ( int i=0;i<n;i++)
cout<<marks[i]<<" ";
29.cout<<" "<<tot<<" "<<avg<<endl;
30.}

```

```

31.void Student::ComputeAvg(int n)
32.{ tot=0;
33.for ( int i=0;i<n;i++)
34.tot +=marks[i];
35.avg=tot/n;
36.}
37.void main()
38.{
39.Student std[60];//std object of
class Student with maximum of 60
students
40.int n; //no of student
41.// Get the number of students in
the class
42.cout<<" Enter number of Students in
the class : ";
43.cin>>n;
44.// get the data for n number of
students
45.for ( int i=0;i<n;i++)
46.std[i].GetData(n);
47.// Compute Average
48.for (i=0;i<n;i++)
49.std[i].ComputeAvg(n);
50.// Print the details of the student
51.cout<<"Roll No"<<" Name"<<"
Marks"<<" Total"<<" Avg"<<endl;
52.for ( i=0;i<n;i++)
53.std[i].DispData(n);
54.}

/* Output :Enter number of Students
in the class : 2

```

```

Enter Students Details
Enter Roll No <space> name : 50595
Ramesh
Enter marks in 2 subjects separated
by spaces :87 89
Enter Students Details
Enter Roll No <space> name : 60695
Gautam
Enter marks in 2 subjects separated
by spaces :99 98
oll No Name Marks Total Avg
0595 Ramesh 87 89 176 88
0695 Gautam 99 98 197 98.5
*/

```

<b>Line s No . 4- 16:</b>	class declaration. Terminated by ;
<b>Line s No .</b>	Member data is declared as private. Name is an array of 20 characters and other variables are double data type.



<b>6– 11:</b>	
<b>Li ne No . 18 :</b>	In void <code>Student::GetData(int n)</code> the symbol <code>::</code> is called scope resolution operator. It tells the compiler to look beyond the controlling braces for the definition in class at line no 13.
<b>Li ne s No . 17 – 25 :</b>	The member function is defined with scope resolution operator
<b>Li ne No . 39 :</b>	<code>Student std[60];</code> // declares 60 objects identified by <code>std[i]</code> of class
<b>Li ne No .</b>	<code>std[i].GetData(n)</code> ; Member functions are always invoked by object using a dot operator

## 6.6 Console IO Operations

Console IO means get input from standard input device, i.e. keyboard and output to standard IO device, i.e. screen or monitor.

### *6.6.1 Console IO Functions – Commands `getchar()` and `putchar()`*

Can be used to input single character at a time from keyboard. Consider the following program, wherein we will read characters into an array and output the array in uppercase.

**Example 6.7: `getchar.cpp` to Demonstrate Usage of `getchar()` and `putchar()`**

---

```
#include<iostream>
using namespace std;
void main()
{
    char city[80]; // declare an array
of 80 characters length
    char c; // we will use it to store
the character input
    int i=0,j=0; // i & j we will use
them as counters
    while ( (c=getchar())!='\n') /*
read in line .'\n' is an end of line
recognized when 'enter' is pressed*/
city[i++]=c;
    //use i as counter once again.
    j=i; // store the value in j. This
is because we will
    i=0; //store it in city[i] and
then post increment i.
    while ( i<j) // output the
character in uppercase
    putchar(toupper(city[i++])); //
post increment i
} // end of main.
/* OUTPUT: : Hi Anand! Welcome to India
HI ANAND! WELCOME TO INDIA */
```

---

Observe that in using while loops we have used brace brackets only for clarity. These while loops have only one line in the body;

hence could have been written without brace brackets as

---

```
while ( c!='\n')  
city[i++]=c;
```

---

### *6.6.2 Console IO Functions – Commands gets and puts*

These commands are used to input and output strings.

#### **Example 6.8: getsputs.cpp to Demonstrate Usage of gets() and puts()**

---

```
1. #include<iostream>  
2. #include<stdio.h> // for gets()  
and puts  
3. #include<string> // for finding  
string length using strlen() function  
4. using namespace std;  
5. void main()  
6. { int len;  
7. char line[80]; // declare an array
```

```

of 80 characters length
8.  cout<<"\nEnter any line: ";
9.  gets(line); // read a line till
'enter' (new line character is pressed )
10. len = strlen(line);
11. puts(line); // output a line
12. cout<<"\nLength of given string =
"<<len<<endl;
13. }
    /*OutputEnter any line: Hello
World
    Hello World
    Length of given string = 11*/

```

---

**L  
i  
n  
e  
N  
o  
.  
2  
:**

includes `stdio.h` for using `gets()` and `puts()` from `stdio` header file. These are imported from C language library. Note that C++ supports all features of C language.

**L  
i  
n  
e**

`#include<string>` gets all the functions for handling strings from a library called `string`.  
`len = strlen(line);` at line No 10

<b>N</b>	computes length of the string using string
<b>o</b>	library.
<b>.</b>	
<b>3</b>	
<b>:</b>	

### 6.6.3 Unformatted Stream IO Functions – *get()*, *put()*, *getline()* **and** *write()*

*Get()* and *put()* are single character input and output functions. They can be used with iostream objects of *cin* and *cout*. For example:

```
char ch ;
cin.get(ch); // fetches a single
character from keyboard and stores it in
ch.
cout.put(toupper(ch)); // converts ch to
upper case and displays on screen
cout.put('z'); // displays z on the
screen
```

In the next example, we will use *get()* and *put* commands. The program will accept lowercase sentence from the user one character at a time till it encounters a full

stop. It will convert each input character into uppercase and displays on the screen.

### **Example 6.9: getput.cpp to Demonstrate Usage of get () and put ()**

```
1.  /* Example 6.10 getput.cpp to
demonstrate the working of get & put.*/
2.  #include<iostream>
3.  using namespace std;
4.  void main()
5.  { int count=1;
6.  char ch;
7.  cout<<"\nEnter a character<. to
stop>";
8.  cin.get(ch);
9.  while( ch!='. ' )
10. { cout.put(toupper(ch));
11. cin.get(ch);
12. count++;
13. }
14. --count; // This is because we
have added for (.) case also
15. cout<<"\n No of characters entered
```

```

= "<<count<<endl;
16. cout<<"\nEnd of Programme"<<endl;
17. }
    /*Output :Enter a character<. to
stop>abcd. ABCD
    No of characters entered = 4
    End of Programme*/

```

---

<b>Line No. 9:</b>	toupper() function converts to uppercase, tolower() converts string to lowercase
--------------------	--

Getline() and write() are line oriented functions. Usage is shown below:

---

```

char text[80];
cin.getline(text,80);// will read white
spaces and \n character.
cout.write(text,80); // out puts the
line of text

```

---

It may be noted that `cout>>text` will not print white spaces. Similarly `cin>> text`



will not read white spaces like tab, space, etc.

### **Example 6.10: `getlinewrite.cpp` to Demonstrate the Working of `getline` & `write` & `write.*`**

```
1. #include<iostream>
2. #include<string>
3. using namespace std;
4. void main()
5. { int len1,len2;
6. char text[20], stg[20];
7. //char line[20];
8. cout<<"\nEnter a line of Text";
9. cin.getline(text,20);
10. len1=strlen(text);
11. cout.write(text, len1);
12. // cout<<"\nNow input put using
cin>>";
13. // cout<<"\nObserve that cin>> can
not read white spaces like space"<<endl;
14. // cin>>line;
15. // cout<<line;
16. cout<<"\noutput second string
```

```

using cin.getline() "<<endl;
17. cin.getline(stg,20);
18. len2=strlen(stg);
19.
cout.write(stg,len2).write(text,len1);
20. cout<<"\nEnd of Programme"<<endl;
21. }

/*Output: Enter a line of TextI
Love India!
I Love India!
output second string using
cin.getline()
I Love Delhi
I Love DelhiI Love India!
End of Programme*/

```

<b>Lines No. 9–11:</b>	show how to use <code>getline()</code> and <code>write()</code> commands with <code>cin</code> & <code>cout</code> .
<b>Lines No. 7 and Lines No. 12–15:</b>	are commented out. You can include the when you want to check the property of <code>&gt;&gt;</code> operator that it cannot read white space.
<b>Line No.</b>	concatenates <code>stg</code> and <code>text</code> .

## 6.7 Summary

1. C++ was developed by Bjarne Stroustrup in 1979 at Bell Laboratories.
2. These additional features include virtual functions, function name and operator overloading, references, free space memory and several features on reusability and extendibility features through inheritance, templates, etc.
3. Exception handling and a well-developed standard template library are two of the advanced features available in C++.
4. C++ allows both single-line comment statements as well as multiline comment statements.
5. Using name space std; is a statement used to resolve conflicts arising due to the same names used in different programs.
6. >> & << are extraction operators in iostream. They are used to stream data in and out with cin and cout, respectively.
7. <<endl is a command directing compiler to go to a new line.
8. Array is derived data type storing the same data type in different contiguous locations in memory.
9. Structure can be used to store different types of data, bundled together as an instance of structure.
10. A class will have member data and member functions. We call this as attributes and behaviour. Once class is created, we will create an instance of class called object.

11. Objects invoke functions and interact amongst themselves to achieve the desired result.
12. Console IO means get input from standard input device, i.e. keyboard and output to standard IO device, i.e. screen or monitor.

## Exercise Questions

### Objective Questions

1. Which of the following are true with respect to C++:

1. Data primacy
2. Process primacy
3. Objective based
4. Object oriented

1. i and iv
2. i, ii and iv
3. i, ii and iii
4. ii and iv

2. `void main()` returns

1. integer
2. 0
3. null
4. void

3. To go to new line the escape sequence required is

1. `'\a'`
2. `'\new'`
3. `'\t'`
4. `'\n'`

4. Function prototype statement will have a semicolon    True/False

5. Function definition statement will have a semicolon    True/False

6. Global declarations and definitions are available to all functions    True/False
7. Type defining a structure would allow shorter names in lieu of key word struct and structure name    True/False.
8. Which of the following are true in respect of C++ declarations?

1. Arrays hold different data types.
2. Array data is stored in continuous memory locations.
3. Structure hold different data types.
4. Class hold data as well as functions.

1. i and ii
2. ii, iii and iv
3. i, ii, iii and iv
4. i and iv

9. Namespace std; in C++ is used for

1. For cin and cout functions
2. For console standard functions
3. For resolving naming conflicts
4. To import standard functions

10. In C++ to declare a variable that does not change its value during program runs, use

1. const declaration before variable declaration
2. global
3. static
4. structure

11. Scope resolution operator is

1. ;
2. :
3. ::
4. (.)

12. Class declaration is terminated always by

1. ;
2. :
3. ::
4. (.)

13. Member functions are invoked by objects using the following operator:

- 1. ;
- 2. :
- 3. ::
- 4. .

14. The operator `>>` can read white spaces like blank line  
TRUE/ FALSE

15. The `cin.getline(string, length)` can read white space  
TRUE/FALSE

16. The `cin.get()` cannot read white spaces  
TRUE/FALSE

#### Short-answer Questions

17. What are global declarations?

18. What does `<iostream>` or `iostream.h` contain?

19. Why are function prototypes declared before `main()` function?

20. Declare an array of integers `X` to hold 25 values. Draw pictorially and put a value of 25 in `X[10]`.

21. Distinguish structure and arrays in C++.

22. Distinguish `get()` and `gets()` commands.

23. Distinguish `getchar()` and `get()`.

24. Declare a structure called `BankCustomer`. Show the fields `name`, `acctno`, `balance`. Type define structure as `customer` and creates an array of customers called `cust` to hold 25 customers of type `BankCustomer`.

25. Use class declaration for problem at 5. Add functions `FindBalance()` and `Transact()` to the class declaration.

26. What are console IO operations?

#### Long-answer Questions

27. Explain C++ program structure and development environment.
28. What are the salient features of the OOP paradigm?
29. What are the special OOP-related features of C++?
30. Explain console IO operations and commands with examples.

### Assignment Questions

31. Write the various steps involved in executing a c program and illustrate this with the help of a flowchart.
32. Write a function program to convert Fahrenheit to centigrade using the formula:  $Celsius = 5 * (fahrenheit - 32) / 9$ .
33. Write a c module to compute simple and compound interest using the formula:  $SI = P * N * R / 100$  and  $CI = P * (1 + R / 100)^N$
34. Write a program to prepare name, address and telephone number of five of your friends. Use the structure called friend.
35. Write a program to store number, age and weight in three separate arrays for 10 students. At the end of data entry, print what has been entered.
36. Using the formula  $A = \text{Squareroot}(s * (s - a) * (s - b) * (s - c))$ , compute the area of the triangle.  $S = (a + b + c) / 2$ , and a, b and c are sides of the triangle.
37. Write a cpp to check if two strings have the same number of characters. Use `getline()`.

### Solutions to Objective Questions

1. a
2. d
3. d
4. True

5. False
6. True
7. True
8. b
9. c
10. a
11. c
12. a
13. d
14. False
15. True
16. False



# 7

## C++ Programming Basics and Control Loops

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to*

- Write C++ programs with good grasp and understanding.
- Understand data types and their usage.
- Understand operators and their precedence and association rules.
- Understand and use control loops.

### 7.1 Introduction

In this chapter, we introduce you to rich C++. We analyse the data types permitted by C++ language, together with various operators like logical operators and arithmetic operators. In Chapter 1, you have of course used these features, but in this chapter we will provide you with the underlying syntax and grammar of C++. We have shown the workings of various types of operators such as binary operators and unary operators and bitwise operators with sample programs. The precedence and association of operators are also presented.

## 7.2 Declaration of Variables

In C++ language, all variables must be declared before they are used. However, C++ gives a luxury of declaring just before you use them. A variable can consist of alphabets and digits. Either upper- and lowercase or mixtures of both cases are allowed. A variable cannot start with a digit. It can start with an `_`. The allowable characters in C++ language are alphabets A

to Z, a to z, numbers 0–9 and the following special characters:

!	*	^	#	%	/	+	%	(	)
-	"		=	{	}	[	]	'	<
>	:	;	,	~	?	&	_	.	BLANK

**Tokens:** A token is an atomic word that is recognized by the compiler and that cannot be broken further. It can be a single character or a group of characters that can be recognized by a C++ compiler.

Examples: Key words, Identifiers, Literals, Punctuations and Operators.

**Keywords:** Reserved and have special meaning in C++ language. A few of the important and commonly used keywords are:

asm	auto	break	case	catch	char	class	const	continue
default	delete	do	double	else	extern	enum	extern	float
for	friend	goto	if	inline	integer	long	new	operator
private	protected	public	register	return	short	signed	sizeof	static
struct	switch	template	this	throw	try	typedef	union	void
volatile	while							

- **Identifiers:** Identifiers are names given to variables, function names, structure names, and so on. The valid variables are: `basic_pay`, `hra`, `FindArea( )`,

d2000k, \_std. The invalid variables and the reasons are:

2found	cannot start with a digit
basic-pay	illegal character
My.pay	illegal character
Your Age	blank space

- **Literals:** They are also called literal constants. They do not change their value during running of the C++ program. C++ allows following literal constants. Integer constants, floating point constants, character constants, string literals, enumeration constants, and symbolic constants.

#### a) Integer Constants:

They can be subdivided into

**Decimal integer constants:** 0 10 –745  
999

- Unsigned integer constants can be specified by appending the letter U at the end. For example, 55556U or 55556u.
- Long integer constants can be specified by appending the letter l s at the end. For example, 789654234L or 7896s.

**Octal integer constants:**

- Only digits between 0 and 7 are allowed. All Octal numbers must start with 0 digit to identify as octal number.

- Allowed octal constants: 0777, 001, 0117, 07565L (octal long)
- Illegal octal constants are: 089 – 8 is illegal, 777 – does not start with 0  
   : -0675.76                    -. is illegal

## Hexadecimal constants:

- A hexadecimal number must start with 0x or 0X followed by digits 0 to 9 or alphabets a to f; both uppercase or lowercase are allowed.
- Allowed hexadecimal constants are: 0xffff, 0xa11f, 0x65000UL.
- Illegal hexadecimal constants are: 0x14.55, illegal character “.”

### b) Floating point constants:

They are base –10 number that can be represented either in exponent form or decimal point representation.

Valid floating point constants are:

–0.01, 789.89765, 5E–5, 1.768E+9

Invalid floating point declarations are:

- 6 invalid . must contain exponent or float point.
- 5E+12.5 Invalid as exponent cannot be float.
- 6,789.00 Invalid character “,”

### c) Character constants:

Character constants can be declared based on the character set followed in a computer. ANSI has standardized these values as shown below:

A	65	a	97	NULL	000
B	66	b	98	LF(line feed)	010
Z	90	z	122	CR(carriage return)	013

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

`'5'` `'X'` `';` `' '`

Note that the character constant `'5'` is not the same as the number 5. The last constant is a blank space. Character constants have integer values known as ASCII values. For example, the statement: `cout<<'a';` would print number 97, the ASCII value of letter a. Since each character constant represents an integer value, it is also possible to perform arithmetic operations on character constants.

Special characters that cannot be printed normally, double quote (`"`), apostrophe (`'`), question mark (`?`), backslash (`\`), etc. can be represented by using *escape sequences*. An escape sequence always starts with `\` followed by special character stated

above. Table 7.1 provides details of escape sequences and special effects.

**Table 7.1** Escape sequences and its special effects

Special Character	Escape Sequence
<i>Bell</i>	\a
<i>Back space</i>	\b
<i>Horizontal tab</i>	\t
<i>Vertical tab</i>	\v
<i>Form Feed</i>	\f
<i>New line</i>	\n
<i>Carriage return</i>	\r
<i>Double Quote</i>	\"
<i>Apostrophe/Single Quote</i>	\'
<i>Backslash</i>	\\
<i>Null</i>	\0
<i>Octal number</i>	\On
<i>Hexadecimal number</i>	\cHn

#### **d) String constants:**

String constants can contain any number of characters in sequence, but enclosed in double quotation marks.

```
"new delhi", "14 Nov 1954", an empty  
string is "".
```

Please note that NULL character `\0` indicates NULL character and is used by C++ language to indicate the end of a string.

#### **e) Enumeration constants:**

Enumeration is a user-defined data type and its members are constants. It can be used effectively to associate integer values to variables. The syntax and example are shown below:

```
Syntax storage class enum variable { var1, var2, var3  
.....};
```

---

```
Examples are: enum bool { FALSE, TRUE};  
enum colors { RED, BLUE, GREEN};  
enum waitque { P0 ,  
P1, P2=5, P6};
```

---

Then integer values assigned with above enum declarations are



---

```
FALSE=0 TRUE=1
RED=0, BLUE=1, GREEN=2
PO=0,P1=1,P2=5,P6=6 and so
```

on

---

We can create instances of enum variable and assign data as shown below:

---

```
color color1,color2;
color1=2; // means color1 will
be GREEN
```

---

In C++ language, enumeration is a list of constant integer values. This enumeration type of declaration is useful when we want to assign constant integer values to names, for example, 0 and 1 to a variable. Consider the example shown below:

---

```
enum bool { no, yes} ; no has a value
0 and yes has a value=1
enum month {jan, feb, mar, apr};
enum color { red , yellow=3 , green };
// red=0 , yellow=3 and green=4
```

---

### **f) Symbolic Constants:**

A symbolic name substitutes a sequence of characters or numerical value that follows it:

---

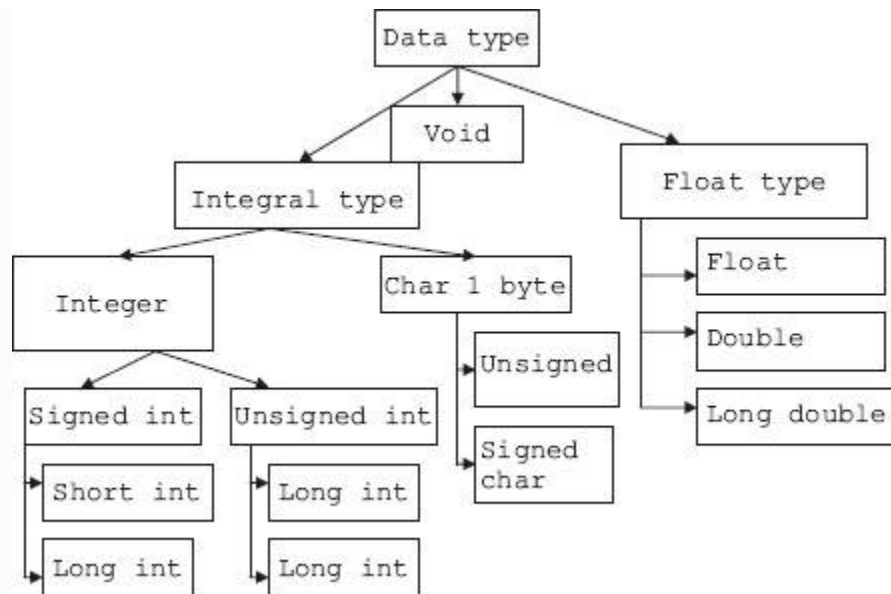
```
# define PI 3.14159
# define MAX 50
# define NAME thunder
```

---

Note that there is no semicolon at the end of the statement.

## 7.3 Data Types

What is a data type? Simply put, it defines a range of permitted values and operations that can be performed on the data type. Data types, also called intrinsic data types, supported by C++ language are shown in [Figure 7.1](#). The ranges allowed for a 32 bit IBM PC and memory requirements are highlighted in [Table 7.2](#).



**Figure 7.1** Data type supported by C++ language

**Table 7.2** Ranges allowed for various data types for a PC

Data type	Size (bytes)	Range allowed	Remarks
Signed char	1	-128-0-127	-A, -d, etc.
Unsigned char	1	0-127	A, B, a, b, etc.
Short signed int	2	-32768 to 0 to 32767	-129, +31560, etc.
Short unsigned int	2	0-65,535	34560, 789, etc
Long signed int	4	-2,147,483,648 to 2,147,483,647	
Long unsigned int	4	0 to 4,294,967,295	
Float	4	-3.4e 38 to +3.4 e 38	
Double	8	-1.7 e 308 to +1.7e 308	Double precision means more bits for significant and exponent.
Long double	10	-1.7 e 4932 to +1.7 e 4932	

Data types can also be distinguished as

1. Intrinsic or basic data types like int, char, float, double, etc. Intrinsic or basic data types are those that do not contain any other data type.
2. Derived data types: array, pointer, etc.
3. User-defined data types: Structure, Unions, etc.

**Derived data types are:** Arrays, functions, pointers, reference and constant.

**User-defined data types are:** Class, structure, union and enumeration.

The smallest, individually addressable memory unit is *byte*. A byte is 8 bits. From Table 2.1, observe that both short int and int have the same memory requirements of 2 bytes. Similarly, an unsigned int will have the same requirements of int. For an ordinary int, leftmost bit is reserved for sign bit. Therefore, balance bits are only 15 and hence range is only  $2^{15}$ , i.e. 32767. Whereas unsigned int complete 16 bits ( 2 bytes ), i.e. a range of  $2^{16} = 65,535$  are possible. Hence, unsigned int will have double the range of ordinary int.

A word about data type called void. It must be clearly understood that void is a data type. It is not “nothing” or NIL or NULL or 0. We can draw an analogy here for better understanding the concept of void. When dealing with gravitational force,  $g = 9.8 \text{ m/sec}^2$  is a state. Similarly,  $g = 0$  or  $-9$  is also a state. Void is a data type which does not belong to any other data types, but is a data type of type void.

## 7.4 Declaration and Assignment Values to Variables

Declaration means mapping the association between the variables and data types.

Following are valid declarations:

---

```
int x , y , z;
float radius=2.56 . //This is
declaration and also assignment of value
to the variable. We can also call this
activity as definition.
float radius[25] ; // declaration of
array of data type float with size 25
double root1=0.3123e-10; // we have
used exponent form.  $0.3123 \times 10^{-10}$ 
short x , y=0 , z; // you can declare
```

```
variables as short or short int
    short int x , y=0 ,z; // Similarly
long int can be declared as long int or
long
    char text[] = "New Delhi"; // The
string contains 9 characters. It will be
stored
    in an array as shown below. Note that
we have left blank for size of the
array.
```

---

**You could also declare specifying the size,  
but size to be correctly specified,**

---

```
taking care of null character as char
text[10] = "New Delhi"
    name of the array : text : N e w D e l
h i \0
    Subscript value : 0 1 2 3 4 5 6 7 8 9
    text[0] contains character N
    text[9] contains null character(\0)
```

---

## 7.5 Expressions

**An expression can comprise any one of the  
following:**

1. A single character or a number.
2. A single constant or a variable.

3. A combination of variables or constants, interconnected by operators.
4. A logical condition that is true (value 1) and false (value 0).

Examples of expressions are:

---

```
root1=(-b+sqrt(d))/(2*a); x=y; ++i;  
x==y; x=0yez;
```

---

## 7.6 Operators

### *7.6.1 IO Operators << and >> , iostream objects cin and cout*

Output operator << directs output stream to standard output device, i.e. display. It is tied to iostream object called `cout`. Similarly, input operator >> gets input from standard input device, i.e. keyboard and tied to iostream object called `cin`.

---

```
Ex: cout<<"\n Enter values of num1 and  
num2 :";  
cin>>num1>>num2;  
cout<<"\n The sum of two numbers =  
"<<num1+num2<<endl;  
Output : The sum of two numbers = sum  
of num1 and num2
```

---

## 7.6.2 Unary Operators

In unary operator, operator precedes a single operand. Unary operators are: `-` , `++` , `--` . `sizeof`. Examples are :

---

```
- 4.0 , -5*(A+B)
++ i , i++ , --i , i-
```

---

`++`, `--` operators are called increment and decrement operators. If they precede the operand, then first the variable is incremented, then the operation is performed. If they follow the operand, then the operation is performed first, and the variable is incremented.

**Example 7.1: Operator1.cpp to Demonstrate Pre- and Postincrement Operators**



---

```
int count = 1;
cout<<count; // output will be 1
cout<<++count;
/* count will be incremented by one and
then operation of
print is performed. Output will be 2.
Now, if you use*/
cout<<count++;
/* count will be printed first. Output
is 2. Then count will
be incremented by 1 to 3.*/
cout<<count; // output will be 3.
```

---

### *7.6.3 Size of Operator*

Will be useful for determining the size allocated for a data type by the computer.

---

```
char city[]="New Delhi";
```

---

### **Example 7.2: Operator1.cpp to Demonstrate Sizeof Operators**

---

```
cout<<"\nSize of integer: "<<
sizeof(int));
cout<<"\nSize of float : " <<
sizeof(float));
cout<<"\nNumber of characters in String
constant city : " << sizeof city ;
Output of above statements would be
Size of integer : 2
Size of float : 4
Number of characters in String constant
city:9
```

---

### **Example 7.3: quadroots.cpp to Obtain Roots of Quadratic Expression**

---

```
/*Example 7.3 quadroots.cpp A program to
compute roots of a quadratic equation.
In this example observe, declaration,
assignment, expressions, math function
This program finds the roots of a
quadratic equation. Formula to compute
the roots are  $(-b + \sqrt{b^2 - 4ac}) / (2a)$ 
and  $(-b - \sqrt{b^2 - 4ac}) / (2a)$  */
```

```

1. #include<iostream>
2. using namespace std;
3. #include<math.h> // for square root
function. h stands for header file
4. // function prototype
5. void FindRoot(float a , float b ,
float c);
6. void main()
7. { // declare three variables as
double precision numbers
8. double a,b,c;
9. // obtain the coefficients
10. cout<<"\nenter coefficients of x^2
, x , and constant:";
11. cin>>a>>b>>c;
12. // Call FindRoot function. We are
sending a , b , c values as arguments
13. FindRoot(a,b,c);
14. }// end of main
15. // function definition
16. void FindRoot(float a,float
b,float c)
17. { double d,root1,root2;
18. d = ((b*b)-(4*a*c));
19. if(d>=0)
20. {      cout<<"\n roots are
real\n";
21.      root1=(-b+sqrt(d))/(2*a);
22.      root2=(-b-sqrt(d))/(2*a);
23.      cout<<"\n
root1="<<root1<<endl;
24.      cout<<"\n

```

```

root2=""<<root2<<endl;
25. }
26. else
27. {      cout<<"\n roots are
imaginary\n";
28.      cout<<"\nroot1="<<-b/ (2*a)
<<sqrt(-d) / (2*a)<<endl;
29.      cout<<"\nroot2="<<-b/ (2*a)
<<sqrt(-d) / (2*a)<<endl;
30. }
31. }// end of function call
/*Output:
Enter coefficiants of x^2 , x ,
and constant:1 -3 2
roots are real
root1=2
root2=1 */

```

---

### 7.6.4 Arithmetic Operators

The basic (also known as intrinsic) operators are

---

+	addition	-	subtraction	*
	multiplication			
/	division	%	modulus (	
	remainder after division).			

---

It may be noted that C++ language does not support exponentiation. Although we will use ^ symbol to denote exponentiation in expression, we have to use a library function call pow to calculate the exponentiation.

***Type conversion:*** If the variable involved in an operation are of different type, then type conversion is carried out before the operation.

If the operation is between a float and double, the float will be converted to double and the result will result in double.

If the operation is between a float or double or long double and a char or int, then char or int will be converted to float or double or long double and the result will result in float or double or long double.

If the operation is between a int and long int, the int will be converted to long int and the result will result in long int.

If the operands are not float or long int, they will be converted to int.

***Type Cast:*** Suppose, we want to declare the result in particular data type, we can

*type cast as shown below:*

---

```
int a , b;  
float x;  
x=(float)a/b; // a/b is integer  
division and the result is converted to  
float.
```

---

### *7.6.5 Relational and Logical Operators*

The relational operators are: > >=< and <=.

These four relational operators have the same precedence. However, they have lower priority than arithmetic operators. The two more operators, known as equality operators == and != have priority just below relational operators.

### *7.6.6 Logical Operators*

These are && and ||. Evaluation of expressions connected by logical operators are done from left to right and evaluation stops when either truth or falsehood is established. In the statement shown below:

---

```
while ( (iflag==0) && ( text[i]!='E')  
)
```

---

`first iflag == 0` is evaluated, if it is true, then only the second expression `text[i] != 'E'` is evaluated. In other words, evaluation stops as soon as truth or false is established.

### *7.6.7 Conditional Expressions: Question Mark Operator*

Suppose you want to allot 10 additional bonus marks to students who put in 100 percent attendance and all others an additional 2 marks. This would result in statements like

---

```
If ( attendance > 100)
    marks += 10; // this is a compound
statement. It means
    Marks=Marks +10
Else
    marks +=2;
```

---

C++ language gives you facility of conditional operator, using which the above 4 lines can be coded as a single line:

---

```
marks = (attendance > 100) ? marks +
10 : marks + 2;
```

---

The syntax is :  $z = \text{exp1} \ ? \ \text{expr} \ 2 \ : \ \text{exp3}$ . Exp1 is evaluated first. If it is true, z is equated to the result of exp2 . Else z is equated to exp3.

### **Example 7.4: Largest.pp to Find Greatest Number Using Conditional Operator**

---

```
/*Example 7.4 Largest.pp to find largest
of three numbers using ternery
operator*/
1. #include<iostream>
2. using namespace std;
3. void main()
4. { int n1,n2,n3,max;
5.   cout<<"Enter three
numbers:"<<endl;
6.   cin>>n1>>n2>>n3;
7.   max=n1>n2?(n1>n3?n1:n3):(n2>n3?
n2:n3);
8.   cout<<"Greatest number is "<<max;
9. }
```



```
/* Output: Enter three numbers:
67 54 98
Greatest number is 98*/
```

---

## Line No. 7:

---

```
max=n1>n2?(n1>n3?n1:n3):(n2>n3?
n2:n3);
```

Firstly  $n1 > n2$  is evaluated.

If  $n1 > n2$

max=( $n1 > n3 ? n1 : n3$ ). If  $n1 > n3$   
returns  $n1$  else  $n3$

Else

max=( $n2 > n3 ? n2 : n3$ ); If  $n2 > n3$   
returns  $n2$  else  $n3$

---

### 7.6.8 Comma Operator

This operator is used to string together several expressions. For example, consider an expression  $x = (y = 5, y + 1)$ . The expression is evaluated from left to right.

---

Firstly,  $y = 5$  hence  $x = 5$ , then  $x = y + 1$ , i.e. 6.

For loop for ( int  $i = 0$ , int  $z = 1$ ;  
 $i \leq 100$ ;  $i++$  ), two statements are  
initialized.

---

## 7.6.9 Bitwise Operators

### Bitwise operators available in C language are

---

& Bitwise AND . Used for masking operation. For example, if you want to mask the first four bits of a number 'n', then we will mask n with a number whose last four bits are 1s, i.e. 0001111. In Octal representation, it is 017. (Remember an octal number starts with 0 and a hexa number starts with 0x.)

```
n = 1 0 0 1 0 1 0 1 =  
149(decimal)  
& 0 0 0 0 1 1 1 1 = 017(octal)  
result n = 0 0 0 0 0 1 1 1
```

Note that the last four bits are 0101 and are unaffected, i.e. they are just reproduced in the result, whereas the left four bits are all 0s, i.e. they are masked.

Bitwise OR. This operator is used when you want to set a bit. For example, we want to set 0th and 2nd bit to 1 for n = 144, then we will use | operator with n and as shown below:

```
n = 1 0 0 1 0 0 0 0 =  
144(decimal)  
| = 0 0 0 0 0 1 0 1 =  
005(octal)  
result n = 1 0 0 1 0 1 0 1 =
```

149(decimal)

^ Bitwise Exclusive OR. Exclusive OR also known as odd function, produces output 1, when both bits are not the same (odd) and produces a 0 when both bits are the same.

n = 1 0 0 1 0 1 0 1 =

149(decimal)

^ = 0 0 0 0 0 1 0 1 =

005(octal)

result n = 1 0 0 1 0 0 0 0 =

149(decimal)

<< Left Shift. Shifting left by one position, bits of a binary number is equal to multiplying the number with 2.

n = 1 0 0 1 0 0 0 0 =

144(decimal)

n<<1 1 0 0 1 0 0 0 0 0 =

288(decimal)

n<<2 1 0 0 1 0 0 0 0 0 0 = 576

>> Right Shift. Shifting right by one position, bits of a binary number is equal to division of the given number with 2.

n = 1 0 0 1 0 0 0 0 =

144(decimal)

n>> 0 1 0 0 1 0 0 0 =

72(decimal)

n>>2 0 0 1 0 0 1 0 0 =

36(decimal)

~ Tilde operator. one's complement operator. This is a unary operator, used

to find one's complement of a given number.

$n = 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1 =$

149(decimal)

$\sim n = 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0 =$  bitwise complement

---

## Example 7.5: bitwise.cpp to Demonstrate the Working of Bitwise Operators

---

```
1. #include<iostream>
2. using namespace std;
3. int main()
4. { int n = 149;
5.   int res;
6.   res = n & 0017;
7.   cout<<"The resultant of Bitwise
AND operator is "<<res<<endl;;
8.   res = n | 0017;
9.   cout<<"The resultant of Bitwise OR
operator is "<<res<<endl;;
10.  res = n && 0017; // this is
logical AND. Truth or false will be
```

output

```
11. cout<<"The resultant of Logical  
AND operator is "<<res<<endl; ;
```

```
12. res = n || 0017; // this is  
logical OR . Truth or false will be  
output
```

```
13. cout<<"The resultant of Logical OR  
operator is "<<res<<endl;;
```

```
14. res = n ^ 0017;
```

```
15. cout<<"The resultant of Exclusive  
operator is "<<res<<endl;;
```

```
16. res = n <<2;
```

```
17. cout<<"The resultant of shift left  
( by 2 bits) operator is "<<res<<endl;;
```

```
18. res = n >>2;
```

```
19. cout<<"The resultant of shift  
right ( by 2 bits) operator is  
<<res<<endl;;
```

```
20. res = ~n;
```

```
21. cout<<"The resultant of NOT  
operator is "<<res<<endl;
```

```
22. return 0;
```

```
23. }
```

```
/*Output : The resultant of  
Bitwise AND operator is 5
```

```
The resultant of Bitwise OR  
operator is 159
```

```
The resultant of Logical AND  
operator is 1
```

```
The resultant of Logical OR  
operator is 1
```

```
The resultant of Exclusive
```

operator is 154

The resultant of shift left (by 2 bits) operator is 596

The resultant of shift right (by 2 bits) operator is 37

The resultant of NOT operator is -150\*/

---

## 7.7 Precedence and Association of Operators

The precedence and association of operators are shown in Table 7.3. The operators at the top have priority more than those that appear later in the table, i.e. operators' priority is highest at the top of the table and lowest at the bottom of the table. Operators on the same line have the same priority.

**Table 7.3** Precedence and association rules for the operators

Operator	Association
----------	-------------

Function call ( ) , [ ] , -> .	Left to right
! ~ ++ -- + - * & sizeof	Right to left
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
= = !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?:	Right to left
= += -= *= /= %= ^= != <<= >>=	Right to left
,	Left to right

- \* / and % have all the same priority
- Unary operators like +, - and \* have more priority than binary operators

## 7.8 Control Loops

An algorithm or a procedure is a sequence of logical steps. Control statements change the sequence of execution of statements as per the requirement of logic. We have already used while for loop and if statements. We will now formalize and consolidate our understanding of control loops.

C++ is a block-oriented language. The program consists of statements. Statements are logically grouped into blocks. Blocks are enclosed in brace brackets. { and } are used to denote start and end of the block in C++. All variables declared inside the controlling braces are called local to the block. This means the value the variable holds is available only inside the block. Statements inside the block are also called compound statements.

Note that there will be no semicolon after the closing brace bracket. For example, in function definition we have enclosed all the statements in brace brackets.

### *7.8.1 Conditional and Branching Statements*



### 7.8.1.1 If Statement

---

The syntax is simple and straightforward

```
if (expression)
    statement;
```

---

### 7.8.1.2 If–Else Statement

---

The syntax is

```
if (expression)
{
    statements;
}
else
{
    statements;
}
```

Else statement is optional. But if used it will be associated with the nearest if statement. In the example shown, else is attached to innermost if.

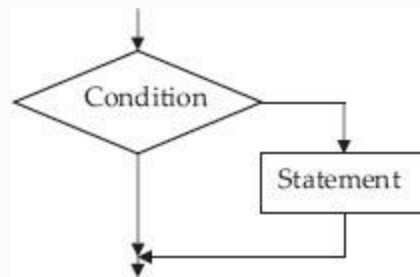
```
    if ( totalMarks > 60)
    if ( total Marks > 70)
        cout<<"passed with
distinction"<<endl;
    else
        cout<<"passed with first
class"<<endl;
```

Use of brace brackets dictate the association rule for else statements.

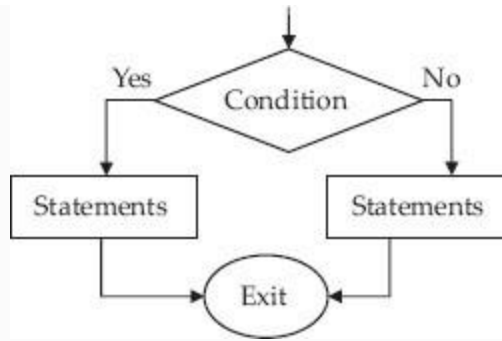
Else in the following code is linked up with first if statement:

```
    if ( totalMarks > 60)
    { if ( total Marks > 70)
      { cout<<"passed with
distinction"<<endl;
      }
    }
```

---



**Figure 7.2** Control flow in if statement



**Figure 7.3** Flow in if–else statement

```
        else // associated with
inner if
        { cout<<"passed with first
class"<<endl;
        }
    }
    else // associated with outer if
    { ...else block.....}
```

## **Example 7.6: FindMax.cpp to Find the Maximum of Two Numbers**

---

```
// Example 7.6: Findmax.cpp to find
maximum of two numbers
1.  #include <iostream>
2.  #include<conio.h>
3.  using namespace std;
4.  long int FindMax (long int a, long
int b);
5.  void main ()
6.  { long int i=150, j=170, k;
7.    long int x=715000, y=918000, z;
8.    k=FindMax(i,j);
9.    z=FindMax(x,y);
10.   cout << "\n Larger of the two
integers : "<<i<<" & "<<j<<" : "<< k <<
endl;
11.   cout << "\n Larger of the two
long integers : "<<x<<" & "<<y<<" : "<<
z << endl;
12.   getch();
13. }
14. //fn definition
15.   long int FindMax (long int a,
long int b)
16.   {
17.     long int max;
18.     //max = (a>b)? a : b;
19.     if ( a>b)
20.       max=a;
21.     else
22.       max=b;
```

```

23.    //max = (a>b)? a : b;
24.    return (max);
25. }

/*Output: Larger of the two
integers : 150 & 170 : 170
Larger of the two long integers :
715000 & 918000 : 918000*/

```

---

<b>Li n es N o. 14 – 2 4</b>	are for function definition. Lines No. 19–20 depict usage of if. Lines No. 21–22 show else statement. Line No. 23 is commented out and uses question mark operator to achieve the same result but with elegance.
--	--

Note that we have included ***conio.h*** for console in and out library. This would facilitate the use of console functions like clear screen (`clrscr()`) and `getch()` for get character operation. Use of `getchar()` would make computer wait for input

through console. It will proceed further only when it receives the input. This feature can be used to observe the result on console. Linux-based compilers like gnu C++ do not support `conio.h`.

### 7.8.1.3 Nested if

When an if statement has another if statement in its body or in the body of else statement we would call it as nested if statement.

#### **Example 7.7: `calcif.cpp` to Simulate Simple Calculator with +, -, \*, / Operators Using Nested Ifs**

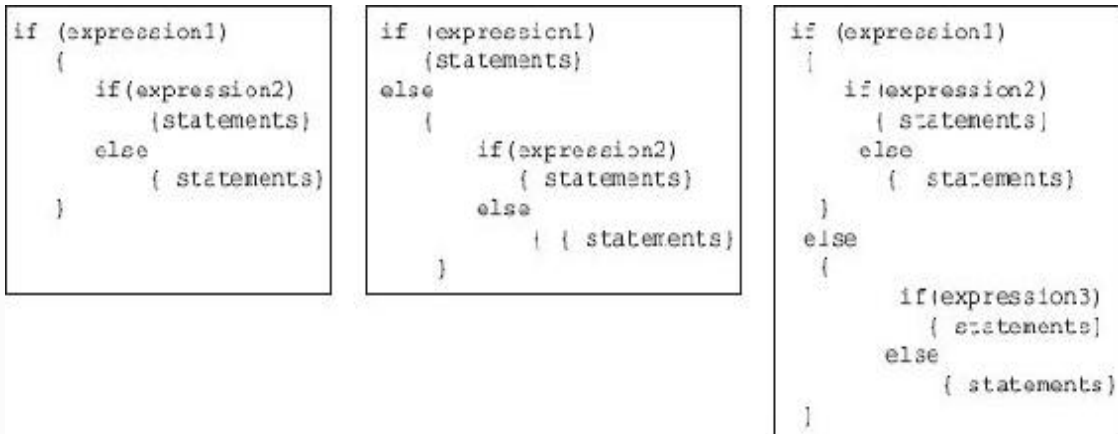
```
/* Example 7.7 program to simulate the
calculator using if-else ladder?*/
1. #include<iostream>
2. #include<conio.h>
3. using namespace std;
4. void main()
5. { char ch; // to store operator
6. float a,b,result;
```

```

7.  cout<<"Enter any two numbers : ";
8.  cin>>a>>b;
9.  cout<<"\nEnter the operation( + ,
- , * , / ) :";
10. cin>>ch;
11. cout<<endl;
12. if(ch=='+')
13.  result=a+b;
14. else
15.  if(ch=='-')
16.  result=a-b;
17.  else
18.  if(ch=='*')
19.  result=a*b;
20.  else
21.  if(ch=='/')
22.  result=a/b;
23.  else
24.  cout<<endl<<"wrong operator";
25. cout<<"\nThe result of " << a <<
ch << b <<" = "<<result;
26. }
    /*Output: Enter any two numbers :
21 18
    Enter the operation( + , - , * , /
) :/
    The result of 21/18 = 1.16667*/

```

---



**Figure 7.4** Types of nested if statement

#### 7.8.1.4 If-Else-If Ladder

Lines No. 12 to 24 depict the if-else-if ladder. It is called ladder because it looks like a ladder. Observe the indentation. Its deep indent and costs in terms of space. Hence the if-else-if ladder can be replaced by the if-else-if, as shown below:

```

if(ch=='+')
result=a+b;
else if(ch=='-')
result=a-b;
else if(ch=='*')
result=a*b;
else if(ch=='/')

```



```
result=a/b;
else
cout<<endl<<"wrong operator";
cout<<"\nThe result of " << a << ch <<
b <<" = "<<result;
}
```

---

### *7.8.2 Switch and Case Statements*

Switch statement evaluates an expression and depending on the numerical value of the evaluation control is branched to corresponding block of statements. The syntax and example problems are shown below:

---

```
switch ( expression)
    case constantexpression1 :
statements1
    case constantexpression2 ;
staements2
    default statements
```

---

**Example 7.8: calcswitch.cpp to Simulate Simple Calculator with +,**

## **-, \*, / Operators Using Switch**

// Example 7.8 program to show the usage of switch and case

```
1.  #include<iostream>
2.  #include<conio.h>
3.  using namespace std;
4.  void main()
5.  {int choice;
6.   int num1,num2;
7.   cout<<"1.Addition"<<endl;
8.   cout<<"2.Substraction"<<endl;
9.   cout<<"3.Multiplication"<<endl;
10.  cout<<"4.Division"<<endl;
11.  cout<<"5.Quit"<<endl;
12.  do
13.  { cout<<"\nEnter your choice: ";
14.    cin>>choice;
15.    switch(choice)
16.    {
17.      case 1:cout<<"\nEnter <num 1&
num2>: ";
18.        cin>>num1>>num2;
19.
20.        cout<<"num1+num2="<<num1+num2<<endl;
21.        break;
22.      case 2:cout<<"\nEnter <num 1&
num2>: ";
23.        cin>>num1>>num2;
```

```

23.    cout<<"num1-num2="<<num1-
num2<<endl;
24.    break;
25.    case 3:cout<<"\nEnter <num 1&
num2>: ";
26.    cin>>num1>>num2;
27.
cout<<"num1*num2="<<num1*num2<<endl;
28.    break;
29.    case 4:cout<<"\nEnter <num 1&
num2>: ";
30.    cin>>num1>>num2;
31.
cout<<"num1/num2="<<(float)num1/num2<<en
dl;
32.    break;
33.    case 5:cout<<"exiting from
program"<<endl;
34.    exit(0);
35.    default: cout<<"Invalid
choice<enter no between 1 and 5
only>"<<endl;
36.    } // end of switch
37. }while(choice!=5);
38. }//end of main
    /*Output
    1.Addition
    2.Subtraction
    3.Multiplication
    4.Division
    5.Quit
    Enter your choice: 3

```

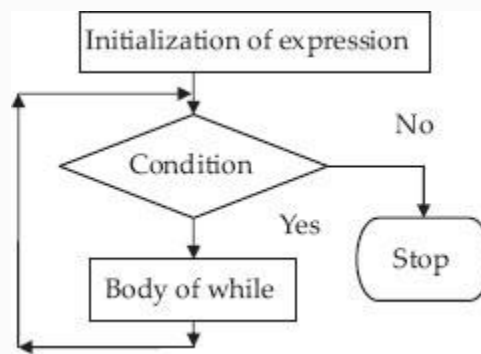
```
Enter <num 1& num2>: 65 2
    num1*num2=130
Enter your choice: 1
Enter <num 1& num2>: 65 2
    num1+num2=67
Enter your choice: 2
Enter <num 1& num2>: 65 2
    num1-num2=63
Enter your choice: 4
Enter <num 1& num2>: 65 2
    num1/num2=32.5
Enter your choice: 5
    exiting from program*/
```

---

Note that switch statement directs flow to the block of statements depending on the integer constant choice. Further it is necessary to separate blocks in a switch statement through ***break*** statement. A break statement, simply put, breaks the nearest brace bracket block, in this case being switch block brace brackets. Also note that the while loop continues till you enter choice of 5. It exits the program through `exit(0)` statement.

### *7.8.3 While Loop*

The while loop is written by a programmer if he is not sure if the while block will be executed. A condition is checked first. If it is true, the while block is executed. Control flow for the while loop is shown in Figure 7.5. The syntax of the while statement is



**Figure 7.5** Control flow for while and for ops

---

*While (expression)//body of while  
contains a single line no brace brackets  
required Statement;*

```
While(expression)  
{ statement1;  
  statement2;  
}
```

```
while(1) // expression is always true  
{ block of statements;
```

```
    } // the loop is called forever while  
loop
```

---

## **Example 7.9: heater.cpp to Use While Loop to Turn off the Heater if Upper Cutoff Temperature has Reached**

The algorithm is: Read the temperature.  
While temperature < upper cut off Switch  
on the heater Else Switch off the heater  
//Heater.cpp

---

```
1.  #include<iostream>  
2.  #include<conio.h> // console input  
    /output for getch() and clrscr()  
3.  using namespace std;  
4.  const int MAX=60;  
5.  void main()  
6.  { float temp ;  
7.      cout<<"Enter <temperature of the
```

```

heater> :";
8.      cin>>temp;
9.      while (temp<MAX)
10.     cout<<"\n Temperature is
lukewarm. Switch on heater"<<endl;
11.     cout<<"\nWater is hot. Switch off
the heater" <<endl;
12. } // end of main
      /*Output :Enter <temperature of the
heater> :67
      Water is hot. Switch off the heater
      */

```

<b>Line No. 4:</b>	declares Max as integer data type and it does not change its value during its run-time.
--------------------	---

## **Example 7.10:   sumwhile.cpp** **//Program to Find Sum of n** **Numbers and their Average Using**

---

```

// Example 7.10 sumwhile.cpp.
1.  #include<iostream>
2.  #include<conio.h>// console input
/output for getch() and clrscr()
3.  using namespace std;
4.  void main()
5.  { int n;
6.  int num ,sum = 0, avg=0,count =1;
7.  // input N
8.  cout<<"\n Enter value of <N>:";
9.  cin>>n;
10. // control loop
11. while (count <= n )
12. {      cout<<"\n Enter value of
"<<count<<"number";
13. cin>>num;
14. sum+=num;
15. count++;
16. }//end of
17. avg=sum/n;
18. cout<<"\n Sum of" <<n<<"numbers
="<<sum<<endl;
19. cout<<"\n Average of"<<n<<"numbers
="<<avg<<endl;
20. getch();
21. } // end of main

/*Output: Enter value of <N>:3
Enter value of 1number34
Enter value of 2number56
Enter value of 3number87

```



```
Sum of 3numbers =177  
Average of 3numbers =59*/
```

---

### *7.8.4 Do-while Loop*

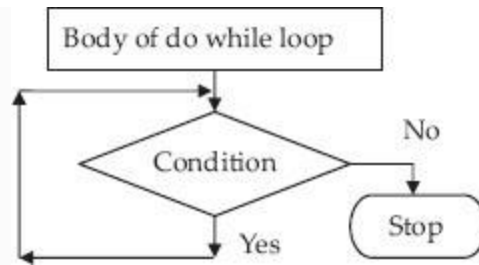
---

The syntax is

```
    Do  
{  
    block of statements  
} while (expression);
```

---

The block is executed first and the condition is checked. If true, the loop is executed till the condition becomes true. We will use do-while loop when we know that the loop needs to be executed at least once, whereas while loop is used when we are not aware if loop needs to be executed or not. The control flow is shown in Figure 7.6.



**Figure 7.6** Control flow for do-while loop

### **Example 7.11: sumwhile.cpp** **//Program to Find Sum of n** **Numbers and their Average Using** **Do-while Loop**

```
1. #include<iostream>
2. #include<conio.h>// console
input/output for getch() and clrscr()
3. using namespace std;
4. void main()
5. { int n;
6.   int num ,sum = 0, avg=0,count =1;
7.   // input N
8.   cout<<"\n Enter value of <N>:";
```

```

9.    cin>>n;
10.   // control loop
11.   do
12.   {   cout<<"\n Enter value of
"<<count<<"number";
13.   cin>>num;
14.   sum+=num;
15.   count++;
16.   }while (count <= n ); //end of do
while. Observe semicolon
17.   avg=sum/n;
18.   cout<<"\n Sum of" <<n<<"numbers
="<<sum<<endl;
19.   cout<<"\n Average
of"<<n<<"numbers ="<<avg<<endl;
20.} // end of main
/*output: Enter value of <N>:3
Enter value of 1number45
Enter value of 2number76
Enter value of 3number78
Sum of 3numbers =199
Average of 3numbers =66*/

```

---

### *7.8.5 For Loop*

For loop, as control loop is used, when we know the exact number of times the loop needs to be executed. The syntax of for loop is

---

```
for ( exp1 ; exp2 ; exp3)
{
block of statements
}
where exp1 is initialization block
      exp2 is condition test block
      exp3 is alter initial value assigned
to exp1
Forever for loop is shown below:
    for ( ; ; )
    {
        statement;
    }
```

---

for loops can be nested. This means that we can write for loop within a for loop. In nested for loop, the inner loop is executed for each value of the outer loop. For example, inner loop is executed for  $i=0$  and the value of  $j$  is varied from 0 final condition. Outer loop is executed for values of  $i$  varying from 0 to  $n - 1$ .

---

The syntax is

```
for ( int i=0;i<n;i++)
{
    for (int j=0;j<n;j++)
    {
```

```
statement
}
}
```

---

### **Example 7.12: sumfor.cpp**

### **//Program to Find Sum of n**

### **Numbers and their Average Using**

### **for Loop**

```
#include<iostream>
#include<conio.h>// console input
/output for getch() and clrscr()
using namespace std;
void main()
{ int n;
  int num ,sum = 0, avg=0,count;
  // input N
  cout<<"\n Enter value of <N>:";
  cin>>n;
  // control loop
  for(count=1; count<=n ; count++)
  {cout<<"\n Enter value of
  "<<count<<"number";
  cin>>num;
```

```
sum+=num;
    avg=sum/n;
    cout<<"\n Sum of" <<n<<"numbers
="<<sum<<endl;
    cout<<"\n Average of"<<n<<"numbers
="<<avg<<endl;
    getch();
} // end of main
/*Output: Enter value of <N>:3
Enter value of 1number67
Enter value of 2number76
Enter value of 3number87
Sum of 3numbers =230
Average of 3numbers =76*/
```

---

## **Example 7.13: nestedfor.cpp A Program to Demonstrate Nested for Loop**

```
#include<iostream>
#include<conio.h>// console input
/output for getch() and clrscr()
using namespace std;
void main()
```

```

{ int i , j ; // counters for outer and
inner for loop
for (i=15; i<16; i++) // outer loop
{
    cout<<"\n Multiplication table for
"<< i<<" X "<<20<<endl;
    for (j=15; j<20 ; j++) // inner loop
    {
        cout<<endl<<i<<" X "<<j<<" =
"<< i*j;
    } // end of inner for loop
} // end of outer for loop
} // end of main
/*Output : Multiplication table for 15 X
20
15 X 15 = 225
15 X 16 = 240
15 X 17 = 255
15 X 18 = 270
15 X 19 = 285 */

```

---

### *7.8.6 When to Use For or While or Do-while*

**For loop** is best suited when programmers have exact knowledge of initialization and final conditions to be met.

---

```

    for ( exp1 ; exp2 ; exp3)
    {
        block of statements
    }

```

```
}  
where exp1 is initialization block  
    exp2 is condition test block  
    exp3 is alter initial value assigned  
to exp1
```

---

The above for loop is equivalent to the following while loop. **While** loop is used when the programmer does not know if at all the while block will be executed. In other words, in a situation where we have to check a condition and then only execute block, we will use while loop.

---

```
exp1;  
while (exp2)  
{    statement;  
    exp2;  
}
```

---

**Do-while loop**, on the other hand, is used when we know that the block is to be executed at least once. In other words, we have to execute the block and then check for a condition.



## 7.9 Break and Continue

### 7.9.1 Break

Break statement is used to exit from the switch control or control loop. We can use break statement to exit from for, while and do-while, and switch control statements. You have already seen the use of break statements in Switch statement. Observe how break is used to come out of if statement below:

#### **Example 7.14: break.cpp Program that Demonstrates Break Statement**

```
#include<iostream>
#include<conio.h>// console input
/output for getch() and clrscr()
using namespace std;
void main()
{
    int count=0;
    int sum,num;
    for ( ;;) // for ever for loop
```

```

    { if(count == 5)
      { cout<<"\n reached upper limit of 5:
breaking the for loop";
        break;
      }
      else
      {
          cout<<"\n Enter value of "
<<count+1<< " number :";
          cin>>num;
          sum+=num;
          count++;
      }
    } // end of for
} // end of main
/*Output
Enter value of 1 number :89
Enter value of 2 number :90
Enter value of 3 number :91
Enter value of 4 number :92
Enter value of 5 number :93
reached upper limit of 5: breaking the
for loop*/

```

---

### *7.9.2 Continue Statement*

Continue is used when we want to stop further processing of loop statements and start at the beginning of the control loop. In the example shown below, we would like to

add 10 points to all odd numbers between 0 to 10 and skip adding to even numbers.

### **Example 7.15: `continue.c.cpp` Program that Demonstrates Continue Statement**

```
#include<iostream>
#include<conio.h>// console input
/output for getch() and clrscr()
using namespace std;
void main()
{ int count;
  for ( count=0;count<=10;count++)
  { if( (count %2)== 0) // the number is
even.
  { cout<<"\n even number: continue at
the beginning if for loop:
"<<count<<endl;
    continue; //control goes to beginning
of for loop
  }//end of if
  else
  {
cout<<"\n odd number : "<<count<<endl;
  }
}
```

```
    } // end of for
  } // end of main
/*output even number: continue at the
beginning if for loop: 0
odd number : 1
even number: continue at the beginning
if for loop: 2
odd number : 3
even number: continue at the beginning
if for loop: 4
odd number : 5
even number: continue at the beginning
if for loop: 6
odd number : 7
even number: continue at the beginning
if for loop: 8
odd number : 9
even number: continue at the beginning
if for loop: 10*/
```

---

### *7.9.3 Goto Statements*

**Goto statements are rarely used due to fears that they would lead to unstructured programs. Goto statements can be conditional and unconditional. The syntax is**

---

```
        goto label //
unconditional branch
```

---

## Example 7.16: goto.cpp to Demonstrate goto Statement

```
#include<iostream>
#include<conio.h>// console input
/output for getch() and clrscr()
using namespace std;
void main()
{ int num ,sum=0,count=0;
  start: cout<<"\n Enter value of
<number> number :";
    cin>>num;
    sum+=num;
    count++;
    cout<<"\n number: "<<num<<endl;
    if ( count > 3)
        goto stop1;
    else
        goto start;
    stop1: cout<<"\n exiting the main
program"<<endl;
} // end of main
/*Output
Enter value of <number> number :55
number: 55
```

```
Enter value of <number> number :67
number: 67
Enter value of <number> number :54
number: 54
Enter value of <number> number :65
number: 65
exiting the main program*/*
```

---

We do not recommend using goto at all. It is always better to get used to while, do-while and for loops for achieving the same result.

#### *7.9.4 Exit Function*

You can force a program to stop whatever it is doing and return the control to operating system by using `exit()` function. For this function you have to include `stdlib.h` in the include section.

---

```
exit(0); // return after successful
completion to operating system(os)
exit(1); // return to os on being
unsuccessful
```

---

## 7.10 Summary

1. A token is a smallest individual unit in a program.
2. C++ has three types of integer constants, namely, decimal octal and hexadecimal constants.
3. Character constants must be enclosed in single quotes. Non-printable character constants are represented by a backslash followed by a character.
4. String constants. String constants can contain any number of characters in sequence, but enclosed in double quotation marks.
5. Enumeration is a user-defined data type and its members are constants. It can be used effectively to associate integer values to variables.
6. Intrinsic or basic data types like int, char, float, double, etc. Intrinsic or basic data types are those that do not contain any other data type.
7. Derived data types are: Arrays, functions, pointers, references and constants.
8. User-defined data types are: class, structure, union and enumeration.
9. Output operator << directs output stream to standard output device, i.e. display. It is tied to `ostream` object called `cout`. Similarly, input operator >> gets input from standard input device, i.e. keyboard and tied to `istream` object called `cin`.
10. Unary operators: In unary operators, operators precede a single operand. Unary operators are: `-`, `++`, `--`, `sizeof`.
11. `sizeof` **operator** will be useful for determining the size allocated for a data type by the computer.
12. **Arithmetic operators.** The basic (also known as intrinsic) operators are `+`, `-`, `*`, `/`, `%`. C++ language does not support exponentiation.
13. **Type conversion:** If the variable involved in an operation are of different types, then type conversion is carried out before the operation.

14. **Type cast:** When we want to declare the result in a particular data type, we can type cast.
15. The relational operators are: > >= < and <= . they have lower priority than arithmetic operators. equality operators = = and != have priority just below relational operators.
16. **Logical operators:** These are && and ||. Evaluation of expressions connected by logical operators are done from left to right and evaluation stops when either truth or falsehood is established.
17. Conditional expressions: Question mark operator. The syntax is: `z = exp1 ? expr 2 : exp3`. Exp1 is evaluated first. If it is true, z is equated to the result of exp2. Else z is equated to exp3.
18. Comma operator: This operator is used to string together several expressions.
19. Loop is executed once first and the condition is checked. Use do-while loop, when we know that the loop needs to be executed at least once.
20. **For** loop, as control loop is used, when we know the exact number of times the loop needs to be executed.
21. **Break.** Break statement is used to exit from the switch control or control loop. We can use break statement to exit from for, while and do-while, and switch control statements.
22. **Continue** is used when we want to stop further processing of loop statements and start at the beginning of the control loop.
23. **Exit function.** You can force a program to stop whatever it is doing and return the control to operating system by using `exit ()` function.

## Exercise Questions



## Objective Questions

1. Which of the following statements are true about variables of C++

1. 0–9 are allowable characters
2. Variable can start with digit
3. Variable can start with underscore
4. ~ tilde is allowable character

1. i and ii
2. i, ii and iii
3. ii, iii and iv
4. i, iii and iv

2. Which of the following statements are invalid variables of C++?

1. My.pay
2. your city
3. basic-pay
4. 2times

1. i and ii
2. i, ii and iii
3. i, ii, iii and iv
4. i, iii and iv

3. Which of the following statements are true about data types of C++?

1. Literals change their values
2. Identifier can start with underscore
3. Void is a data type
4. In C++ 1 means true and 0 means false

1. i and ii
2. i, ii and iii
3. ii, iii and iv
4. i and iv

4. Which of the following are legal and allowed in respect integer constants?

1. Octal constants:089

2. 0xa22f
3. 0x14.55
4. 5E+12.5

1. i and ii
2. i, iii and iii
3. i, ii and iv
4. i and iv

5. For modulus operator both operands must be

1. Float and integer
2. Float and double
3. Integers
4. int & double

6. What is the output of the following code: `int a, b=3; a = b; a+=2; // equivalent to a=a+2`

1. 4
2. 5
3. 6
4. 3

7. What is the output of following code?

---

```
int a,b,c; a=2; b=7; c = (a>b) ? a
: b;
```

---

1. 4
2. 5
3. 6
4. 7

8. Which of the following are true with respect to IO operators?

1. >> is called extraction operator
2. << is tied to cout
3. cin and cout are objects of iostream
4. >> can read white spaces

1. i, ii and iii
2. i, iii and iv

- 3. i, ii and iv
- 4. i and iv

9. Which of the following are true with respect to precedence and association of operators?

- 1. + – have left to right association
- 2. \* / and % have all the same priority
- 3. \* / % association is from right to left
- 4. Unary operators have less priority than binary operators

- 1. i and ii
- 2. i, iii and iv
- 3. i, ii and iv
- 4. i and iv

10. The following control loop is preferred when loop needs to be executed at least once:

- 1. While
- 2. Do-while
- 3. For loop
- 4. Switch

11. The following control loop is preferred when not sure about loop execution:

- 1. While
- 2. Do-while
- 3. For loop
- 4. Switch

12. The following control loop is preferred when initial and final conditions are known:

- 1. While
- 2. Do-while
- 3. For loop
- 4. Switch

13. Break statement takes the control to

- 1. Goes out of the innermost loop that contains the break statement
- 2. To the beginning of the program
- 3. To the end of the program
- 4. Goes out from all the nested loop

14. Continue statement takes the control to

1. To the bottom of the loop
2. To the beginning of the loop
3. To the next statement
4. To the end of the program

15. `exit()` function returns control to operating system. TRUE/FALSE

#### **Short-answer Questions**

16. What are the basic data (fundamental, intrinsic) types of C++?

17. What are the derived data types supported by C++?

18. What are the user-defined data types supported by C++?

19. How are enumeration data types useful?

20. Explain `sizeof()` operator.

21. Distinguish `&` operator and `&&` operator.

22. Give the syntax of `?` operator.

23. How is comma operator used?

#### **Long-answer Questions**

24. What are the data types allowed by C++? Explain with examples.

25. What are IO operators? Explain with examples.

26. Explain type conversion and type casting of operators.

27. Explain bitwise operators with examples.

28. Explain precedence and association of operators.

29. Distinguish the switch and if else statements.

30. When do you use for statement and while statements? State the situation when for statements are better than while statements.

#### **Assignment Questions**

31. Candidates have to score 90 or above in the IQ test to be considered eligible for taking further tests. All the candidates who do not clear the IQ test are sent reject

letters and others are sent call letters for further tests.  
 Represent the logic for automating this task.

32. Write C++ code for following series:

1.  $1+3+5+7+\dots+n$
2.  $1+x^2+x^4+x^6+\dots$  n terms
3.  $(1-x)^n = 1 + nx + ((n(n+1)x^2)/1.2) + ((n(n+1)(n+2)x^3)/1.2.3) + \dots$

33. Write a C++ program to display the ascii table.

34. Write a program to read a line from the keyboard and print the line using a suitable encryption. Simple encryption can be a substitution with next character A with B, a with b, and z with a and so on. De-crypt and display the original message.

### Solutions to Objective Questions

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

# 8

## Functions, Storage Class Preprocessor Directives, and Arrays and Strings

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to*

- Understand functions and function templates and how C++ handles function execution.
- Understand the memory mapping adopted by C++ and storage classes.
- Program C++ with the help of macros and preprocessor directives.
- Understand the concepts and use array and string data structure.

## 8.1 Introduction

It is common knowledge that experts who are dedicated to a particular job will perform the job better and faster. We approach architects for making functional and aesthetic housing. Similarly, we hand over control to doctors to take care of our health. In C++ language, we hand over jobs to functions specially written for the purpose. Similarly, if we can make a function perform more than one task depending on the requirements of the user, then such programs are not only efficient but are also versatile. We can use function templates, so that a single function can handle all types of data.

This chapter also introduces you to the C++ way of handling storage and memory. The variables are stored based on the storage specifiers we use while declaring the variable. These are called storage classes. The storage in turn decides the access times. Preprocessor directives are directives issued to compilers for efficient execution of programs. This chapter introduces the

reader to concepts involved in these directives. Arrays and strings are derived data types and most widely used features of C++. Several problems are included in examples and exercise problems to fortify the concepts involved. From now on, we print line numbers only when we have something new to tell you.

## 8.2 Why Use Function Templates?

The idea is to divide the main problem into smaller modules and write functions to implement the modules. The main function in C++ language is called: `main()` .

Interestingly, it is also a function, albeit a main function. `Main()` function calls other functions. Calling function is one which calls a called function. The function prototype is declared before, `void main()` , in the global area, so that it is accessible by all other functions. The syntax is:



```
ReturnType    Function name  ( Argument List) ; // note the semicolon
  ↙           ↙             ↙
int      FindMax      ( int a , int b , int c );
```

void main () calls the function FindMax () by supplying the arguments and receives the result through return type.

---

```
ans = FindMax ( a, b ) ; // a,b,
and ans are all integer data types
```

Function Definition is given after main program as

---

```
int FindMax ( int a , int b , int
c) // note the absence of semicolon
{ // Function Code here }
```

Functions handle data. There are several intrinsic data types supported by C++ like int, float, double, etc. Suppose we need a function to sort the given data. The data could be data of integers or data of floats or data of double. Do we write a separate function for each of the data types or is there

a tool or method wherein one function takes care of all data types. Function template is the solution provided by C++. The format for declaring function templates with type parameters is:

---

```
template <class identifier>
function_declaration;
template <typename identifier>
function_declaration;
```

---

For example, we could declare a function template for finding a maximum of two numbers as

---

```
Template<class
T>
T FindMax( T a
, T b);
```

---

Below the main function, we could define the template function as

---

```
Template <class T>
```

```
T FindMax(T a , T b) { return ( (a>b)
? a : b ) ; }
```

---

While at the time of declaration or definition, we do not know the data type of T, we can use it as data type T and forward data type as parameter. We would call the function as follows:

---

```
function_name <type> (parameters);
FindMax <int> (x,y); // x & y are
data type integers
```

---

### **Example 8.1: A Template Function Based cpp for Computing the Maximum of Two Numbers.**

#### **Function Template: FindMax.cpp**

```
#include <iostream>
using namespace std;
//fn template definition
```

```

template <class T>
T FindMax (T a, T b);
void main ()
{ int i=50, j=70, k;
  long int x=75000, y=98000, z;
  k=FindMax<int>(i,j);
  z=FindMax<long int>(x,y);
  cout << «\n Larger of the two integers
: «<<i<<» & «<<j<<» : «<< k << endl;
  cout << «\n Larger of the two long
integers : «<<x<<» & «<<y<<» : «<< z <<
endl;
  }
  //fn template definition
  template <class T>
  T FindMax (T a, T b)
  {T max;
  max = (a>b)? a : b;
  return (max);
  }
  //Output :Larger of the two integers :
50 & 70 : 70
Larger of the two long integers: 75000 &
98000 : 98000*/

```

---

A function template can accept two parameters of different types and return value of any one of the parameters. Consider the declaration

---

```
long int a ; int b;  
template< class S , class T>  
T FindMax( S a , T b) { return ( ( a>b)? a: b ); } . The compiler will do  
necessary type conversions and return  
data type of T
```

---

### 8.3 Call by Value

Mode of data transfer between calling function and called function, by copying variables listed as arguments into called function stack area and, subsequently, returning the value by called function to calling function by copying the result into stack area of the main function is called ***Call by value***. Note that as copying of the variables, both for forward transfer and return transactions, are involved, it is efficient only if values to be transferred are small in number and of basic data type.

## **Example 8.2: swap.cpp. A Program to Demonstrate Concepts of Functions So Far Discussed and also Bring Out Concept of Pass by Value**

```
#include<iostream>
using namespace std;
// function prototype declarations
template<class T>
void Swap ( T x , T y); // to
interchange values
void main() //Calling Function
{ float x=100.00; // x & y are local
to main()
float y=1000.00;
int a=5 , b=10;
cout<<"\n Before calling Swap function
<x and y >"<< x<<"\t"<<y<<endl;
cout<<"\n Before calling Swap function
<a and b >"<< a<<"\t"<<b<<endl;
// call the function and pass
arguments x & y
Swap(x,y); //Called Function
Swap(a,b); //Called Function
```

```

    cout<<"\n After return from Swap <x
and y >"<<x<<"\t"<<y<<endl;
    cout<<"\n After return Swap function
<a and b >"<< a<<"\t"<<b<<endl;
    }//end of main
// function definition
template<class T>
void Swap ( T x , T y)
    { float temp ; // temp is local to
Swap function
    temp = x; // store x in temp
    x=y; // store y in x
    y=temp; // store temp in y
    cout<<"\n Inside Swap <x and y
>"<<x<<"\t"<<y<<endl;
    } // end of Swap//
    //output :Before calling Swap function
<x and y >100 1000
    Before calling Swap function <a and b
>5 10
    Inside Swap <x and y >1000 100
    Inside Swap <x and y >10 5
    After return from Swap <x and y >100
1000
    After return Swap function <a and b >5
10*/

```

---

There are two important lessons to be learnt here. Firstly, notice that **function template** for **Swap function** has handled

for two sets of data, i.e. **x and y of float data type** and **a and b of integer data type**. That is the power and utility of function template. Secondly, observe the output. While the function Swap has done its job of interchanging the values of x and y, as shown by cout statement inside Swap in the main program, the values did not interchange. What does this mean? The answer lies in the fact that variable values and operations you do on them are local to block, i.e. local to function Swap. It has no connection with variables x and y belonging to the `main()` function.

Indeed, `main()` function passes arguments by copying these values into stack area of the function Swap. It is like passing a xeroxed document and retaining the original. Obviously, the changes you make on the xerox copy will not be reflected on the original.

## 8.4 Call by Reference

For large data items occupying large memory space like arrays, structures and



unions, overheads like memory and access times become very high. Hence, we do not pass values as in call by value; instead, we pass the address to the function. Note that once function receives a data item by reference, it acts on data item and the changes made to the data item also reflect on the calling function. This is like passing address of original document and not a xerox copy. Therefore, changes made on the original document applies to the owner of the document as well. Let us see how our Swap program behaves when we pass the variable by reference, i.e. by passing address of variable, as we intended by reflecting the changes made in the function in the calling function as well.

**Example 8.3: RefSwap.cpp to Demonstrate the Concept of Passing of Variable by Reference**

---

```
#include<iostream>
using namespace std;
// function prototype declarations
template<class T>
void Swap ( T & x , T & y); // to
interchange values by reference
void main() //Calling Function
    { float x=100.00; // x & y are local
to main()
    float y=1000.00;
    int a=5 , b=10;
    cout<<"\n Before calling Swap function
<x and y >"<< x<<"\t"<<y<<endl;
    cout<<"\n Before calling Swap function
<a and b >"<< a<<"\t"<<b<<endl;
    // call the function and pass
arguments x & y
    Swap(x,y); //Called Function
    Swap(a,b); //Called Function
    cout<<"\n After return from Swap <x
and y >"<<x<<"\t"<<y<<endl;
    cout<<"\n After return Swap function
<a and b >"<< a<<"\t"<<b<<endl;
    }//end of main
// function definition
template<class T>
void Swap ( T &x , T &y)
    { float temp ; // temp is local to
Swap function
    temp = x; // store x in temp
```

```

    x=y; // store y in x
    y=temp; // store temp in y
    cout<<"\n Inside Swap <x and y
>"<<x<<"\t"<<y<<endl;
    } // end of Swap//
//output : Before calling Swap function
<x and y >100 1000
    Before calling Swap function <a and b
>5 10
    Inside Swap <x and y >1000 100
    Inside Swap <x and y >10 5
    After return from Swap <x and y >1000
100
    After return Swap function <a and b
>10 5*/

```

---

For group and user-defined data structures, like structures and unions, call by value method is inefficient. Instead, we will employ call by reference method to move large data items. Study the following example. **Sort.cpp**. A program to demonstrate passing of array by reference. In this example, we will pass the array of integers by reference to a function called `intsort`, which sorts the array in the ascending order.

---

## Example 8.4: Sort.cpp. A Program to Demonstrate Passing of Array by Reference

```
#include<iostream>
using namespace std;
// function prototype declarations
template<class T>
void Sort( int n , T a[]); // receives
number of items & array
void main()
{ int i,n=10;// number of items
  int a[] ={
67,87,56,23,100,19,789,117,6,1}; //
array with 10 elements
  double b[] = {
18.0,87.9,92.1,1.0,5.0,8.7,78.99,34.0,21
.5,10.0};// array of double
  cout<< "\n Given integer array"<<endl;
  for (i=0;i<n;i++)
    cout<<" "<<a[i];
  // we are passing array a. Note that
array name is 'a' . Name is address.
  Sort(n,a);
  cout<<"\n Sorted integer array \n";
  for (i=0;i<n;i++)
```

```

        cout<<" "<<a[i];
    cout<< "\n Given double array"<<endl;
    for (i=0;i<n;i++)
        cout<<" "<<b[i];
    // we are passing array b. Note that
    array name is 'b' . Name is address.
    Sort(n,b);
    cout<<"\n Sorted double array \n";
    for (i=0;i<n;i++)
        cout<<" "<<b[i];
} //end of main
// function definition
template<class T>
void Sort(int n , T a[] )
{ for ( int i=0; i< n-1; i++) // last
value need not be sorted
    { T temp;
        // find the smallest of remaining
        numbers and exchange it with for ( int
        j=i+1; j< n; j++)
            { if (a[j] < a[i])
                { // swap
                    temp=a[i];
                    a[i]=a[j];
                    a[j]=temp;
                }
            }
    }
}

//Output : Given integer array
        67 87 56 23 100 19 789 117 6 1
Sorted integer array

```

```
1 6 19 23 56 67 87 100 117 789
Given double array
18 87.9 92.1 1 5 8.7 78.99 34 21.5 10
Sorted double array
1 5 8.7 10 18 21.5 34 78.99 87.9
92.1*/
```

---

As array is a data structure comprising several data types, compiler passes it as reference. Note that we have called `Sort(n, a);` 'a' means name of the array. Name means the address of the array. In effect, we have given the address of the array to Sort function. The algorithm we have used for sorting integers is simple, though not very efficient. We will learn more about call by reference through usage of pointers.

## 8.5 Call by Constant Reference

When you pass a variable by reference, because of its large size, there is a danger that receiving function can alter the value. If you want to prevent the function from

changing the value, there is a provision to pass by constant reference.

### **Example 8.5:**

#### **callbyconstref.cpp: Passing Constant Reference to a Function**

```
#include<iostream>
using namespace std;
// function prototype declarations
void PrintNetPay(const float & BP ,
float & cr , float db); void main()
{ float bp=2000.00, cr = 0.60*bp, db =
.25*bp;
    // we are passing variables by value ,
by reference and by constant reference
    PrintNetPay(bp,cr,db);
} //end of main
// function definition
void PrintNetPay(const float &bp, float
& cr , float db)
{ float total = bp+cr-db;
    //bp+=1000.00; // error . l-value
specifies const object. constant
reference .
```

```
//Hence cannot be altered. Hence  
commented out  
cout<<"\n Total Net Pay = "<<  
total<<endl;  
}  
//Output :Total Net Pay = 2700 */
```

---

## 8.6 Recursion

Recursion is an advanced feature supported by C++ language. Recursion means a function calling itself. Consider the problem of finding a factorial of a number. In the example, we can clearly see that `Fact (5)` is calling `Fact (4)` and so on.

---

```
Fact(5) = 5 x 4 x 3 x 2 x 1  
        = 5 x Fact(4)  
        = 5 x 4 x Fact(3)  
        = 5 x 4 x 3x Fact(2)  
        = 5 x 4 x 3x 2 x Fact(1)  
        = 5 x 4 x 3x 2 x 1 x Fact(0)  
        = 5 x 4 x 3x 2 x 1 ( Fact(0) =1)
```

---



## Example 8.6: factrecur.cpp A Program for Finding Factorial of a Number Using Recursion

---

```
#include<iostream>
using namespace std;
// function prototype
int fact(int a);
void main()
{ int x,ans;
  cout<<»enter a number:»;
  cin>>x; //input from the user*/
  ans=fact(x); //function call*/
  cout<<»\nfactorial=»<<ans<<endl;
} //end of main*/
int fact(int a) //function definition*/
{   int ans;
    if(a<=0)
        return(1); // fact(0) = 1 by
definition
    else
        ans =a*fact(a-1); //function call
with recursion*/
    return(ans); //returns the value of
fact to main*/
```

```
} //end of function fact*/  
// Output: enter a number:7  
factorial=5040*/
```

---

## 8.7 Inline Functions

Each time a function is called there are some overheads like saving the current address, branching off to code area where function code area, passing arguments and returning to main function on completion of execution of function. For large functions, this process is justifiable, but for one or two line body functions, this lengthy and resource-consuming process is not justifiable. Hence, C++ has provided inline function.

---

```
template<class T>  
inline void FindMax( T & a , T & b ){  
    return ( (a > b) ? a : b ) ; }
```

---

When compiler sees the keyword inline, it places the entire code for FindMax rather than resorting to its usual function-handling technique, thus saving the overheads associated with C++ way of handling

functions. Caution: Using inline functions for large functions can be counterproductive as entire functions are reproduced for each call of the function. Therefore, use inline functions for one or two line functions with short codes. Functions declared and defined inside the class definition are inline functions automatically, though you might not have used the keyword inline.

## 8.8 Function Overloading

When a single function can do more than one job, then again overheads of the compiler get reduced. For example, if one wants to compute the area of a circle, the surface area of a football as well as the surface area of a cylinder. How many functions does one have to write? Normally, three functions. But in C++ one function overloaded to perform all the three jobs is sufficient. Overloaded functions decide which version of the code is to be loaded into primary memory depending on the argument supplied by the user. Why is overloading important? Functions have to

be compiled and loaded into primary memory. If a function is NOT overloaded, all the functions have to be loaded and linked at compile time itself. The user may or may not use all the functions loaded, thus wasting primary memory and making it unavailable to solve complex problems requiring more primary memory

---

```
// one argument area of circle
template<class T>
inline void FindArea( T & r ){ return (
2*3.14158*r*r ) ; }
// two argument surface area of the
cylinder
template<class T>
inline void FindArea( T & r , T &h ){
return ( 4*3.14158*r*h ) ;
}
```

---

## 8.9 Default Arguments

Functions accept arguments and perform their job. But what happens if you do not supply the values for the arguments? There is a provision in C++ to specify the default option. If you supply the arguments, the function would adopt them; if you do not

supply the values for the arguments the function will assume default options. You can specify the default option for all or only for part of the arguments, but specifying default options must commence from left to right. It is illegal (compiler error) to specify default values for right edge arguments.

---

```
float FindVolume( float l=10.0, float
b=20.0 , float h=30.0); //
default for all . O.K.
float FindVolume( float l=10.0, float
b=20.0 , float h); // default
only for l & b. O.K.
float FindVolume( T l=10.0, T b , T h);
// default only for l . O.K.
float FindVolume( float l, float b ,
float h=30.0); // default only
for h . Not O.K.
```

---

We must supply default arguments from left to right. You cannot specify default value for h without specifying values for l and b. In the example program that follows, we will show the working of inline functions, function overloading and default functions. The problem is to find the area of different objects like circle, cylinder, etc. The

function, `FindArea()` will be overloaded to achieve area of square or area of rectangle, depending on whether the user supplies one argument or two arguments. Of course, we will use inline function templates as usual for practice:

### **Example 8.7: `fnoverload.cpp` A Program to Show fn Overloading**

```
#include<iostream>
using namespace std;
// function prototype declarations
template<class T>
    inline T FindArea( T r ){ return (
r*r ) ; }
    // two argument surface area of the
cylinder
template<class T>
    inline T FindArea( T r, T h ){ return
( r*h ) ; }
    inline int FindSum( int a=10 , int
b=20 , int c=30) { return a+b+c ;}
void main() //Calling Function
{cout<<"\n Function overloading for
```

```

Different Number and Different Types of
Arguments"<<endl;
cout<<"\n calling FindArea(r) function <
r >-integer argument:"<<FindArea(10)
<<endl;
cout<<"\n calling FindArea( r , h )
function - float 2 arguments
:"<<FindArea(10.5,20.5)<<endl;
cout<<"\n calling FindArea( r , h )
function - integers 2 arguments
:"<<FindArea(10,20)<<endl;
cout<<"\n*****"<<endl;
cout<<"\n a and b and c values set as
default values are
:"<<10<<"\t"<<20<<"\t"<<30<<endl;
cout<<"\n FindSum with no arguments
supplied " <<FindSum()<<endl;
cout<<"\n FindSum with : a (one )
arguments supplied
"<<100<<"\t"<<FindSum(100)<<endl;
cout<<"\n FindSum with : a and b (two )
arguments
supplied"<<100<<"\t"<<200<<"\t"
<<FindSum(100,200)<<endl;
cout<<"\n FindSum with : a and b (two )
arguments
supplied"<<100<<"\t"<<200<<"\t"<<300<<"\
t"<<FindSum(100,200,300)<<endl;
}
//Output: Function over loading for
Different Number and Different Types of
Arguments

```

```
calling FindArea(r) function < r >-  
integer argument:100  
calling FindArea( r , h ) function -  
float 2 arguments :215.25  
calling FindArea( r , h ) function -  
integers 2 arguments :200  
a and b and c values set as default  
values are :10 20 30  
FindSum with no arguments supplied 60  
FindSum with : a (one ) arguments  
supplied 100 150  
FindSum with : a and b (two ) arguments  
supplied100 200 330  
FindSum with : a and b (two ) arguments  
supplied 100 200 300 600*/
```

---

## 8.10 Memory Management of C++

The following table gives details of memory management of C++. It is important that you clearly understand the concept.

Memory can be broadly divided into data area or code area also called data segment and code segment. Based on the requirements of access times, scope and life of variables, the memory is divided into global, heap, code area, stack and static area, as shown in Table 8.1.



**Table 8.1** Memory organization of C++ language

<b>Global:</b> Include section, function declarations, structures and global variables	All declarations and definitions are accessible to all the functions.
<b>Heap Memory:</b> Access to heap memory is only through pointers i.e. through New and delete functions	Memory remaining after allocating space to global, code and stack/static variables
<b>Code Area:</b> The C++ code, Functions etc. are stored in this section	Code for all functions is stored here
<b>Stack Variables:</b> All variables you declare are stores on Stack. Also called Stack variables	Life of variable is till life of function in which it is declared, called local variables
<b>Static Variables:</b>	Life of static variable is till of <code>main ()</code> . Accessible to all Functions

**Global Variables:** These are declarations such as include sections, define statements, function prototype declaration, and declarations of structures and unions, which have scope that extends to all

functions. It means that any variable declared in global section can be accessed by all functions. This section appears before the `main ()` function.

***Code Area:*** All the functions and their codes are stored here.

***Stack Area:*** All the variables declared in a function are automatically allocated space in the stack area of the memory. The scope of the variable is local. This means that a variable declared in a function is accessible only within the function. Further, the life of variables declared within the function is life of the function itself, i.e. within the brace brackets of the function. Hence, these variables are called local variables or automatic or autovariables. Consider the example given in Table 8.2 for local or autovariables.

**Table 8.2** Storage classes with their location, scope and life span

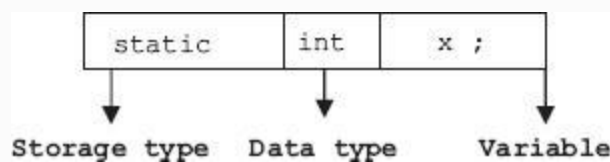
Storage class	Location	Scope	Life
a) Automatic or auto or stack storage	stack	local	within the block { }
b) Register storage	registers	local	within the block { }
c) Static storage	static	local	within the block { } value stays even after termination of function.
d) External	global before main()	entire program	When all functions need the value declare it as external. Till main ()

***Static Variables:*** In several situations, you will need accessibility of variables declared in a function for all function; we will declare such variables as **static**. For example,

<pre> Void main() {     float x=100, y=200,z;     float a[] = { 10.0,20.0,30.0};     .....     z=FindArea(x,y);     cout&lt;&lt;\n area ="  , z); } // end of main </pre>	<pre> float FindArea( float x , float y) { static int times=0; float ans;     float ans;     ans=x*y;     return ans; } // end of FindArea </pre>
<p>x, y, z and array a are local to main()  They are stored on stack area  x &amp; y are passed as arguments to FindArea()  FindArea copies the answer onto z  Life of variables is life of void main()</p>	<p>ans is local to FindArea()  return ans; statement copies ans in to z belonging to main()  Z is printed as output</p>

Variables declared as static are stored in contiguous memory locations. The syntax is shown in Figure 8.1. The only special feature

of static area is that the variables in this static area are not erased, even if function terminates. For example, we have declared times as static int to keep track of the number of times FindArea is called by main() function. We have also initialized times = 0. It is incremented when the first time function is called. Next time, when FindArea() is called it is incremented by 1 to 2.



**Figure 8.1** Syntax for static variables

***Heap Memory or Free Space:*** After allocating mandatory spaces for all the above storage classes, the memory still free is called heap memory or free space. This is dynamic memory available to the programmer for requirements of memory at

run-time. But access to this space is available only through pointers.

### *8.10.1 Types of Storage Classes*

Storage classes supported by C language are shown in Table 8.2.

Note that while static global and external declarations have reach to all functions, there is one vital difference. Static global declarations and definitions are available only through a single `main()` function, whereas declarations and definitions using `extern` are available even to programs written separately with different file names but compiled and linked with the `main()` program. An example will make the concepts clear:

**Example 8.8: `stackstatic.cpp` A Program to Demonstrate Usage of Static and Stack(auto) Variable**

---

```
#include<iostream>
Using namespace std;
#include<conio.h>
float FindArea ( float x , float y); //
function prototype declarations
void main()
{ int i;
  float x=200, y=100, area=0,ans;
  for (i=0;i<10;i++) // call function
FindArea 10 times
  { ans = FindArea(x++,y++);cout<<"\n
area = " << ans;}
  cout<<»\n Address of variable ans in
the main function : "<<ans;
  }// end of main
// function definition
float FindArea(float x, float y)
{ static int times=0; // static variable
to keep track of number of times
// function is called.

  float ans;
  ans=x*y;
  times++;
  cout<<"\n number of times we have
called findArea function = "<<times;
  if (times ==10)
    cout<<»\n Address of variable ans in
the FindArea function : "<<ans;
    return ans;
  }// end of FindArea
/*Output : number of times we have
```

```
called findArea function = 1
  area =20000.00
  number of times we have called findArea
function = 2
  area =20301.00
  number of times we have called findArea
function = 3
  area =20604.00
  number of times we have called findArea
function = 4
  area =20909.00
  number of times we have called findArea
function = 5
  area =21216.00
  number of times we have called findArea
function = 6
  area =21525.00
  number of times we have called findArea
function = 7
  area =21836.00
  number of times we have called findArea
function = 8
  area =22149.00
  number of times we have called findArea
function = 9
  area =22464.00
  number of times we have called findArea
function = 10
  Address of variable ans in the FindArea
function : 12ff04
  area =22781.00
```

```
Address of variable ans in the main  
function : 12ff6c*/
```

---

**Example 8.9: `reg.cpp`. A Program to Demonstrate Usage of Register Storage Class `usage.Stack(auto)` Variable**

Note that registers of CPU reside on the processor chip; the exact number depends on the hardware of the processor and is considered the primary memory and hence enjoy fastest access times. However, having limited memory, they have to be used sparingly. For example, variables which are most frequently used by CPU can be declared as register variables thus saving access times. In the following example, we would use register memory to store counter of control loop, square and square root of a number.



---

```
#include<iostream>
using namespace std;
void main()
{register counter,square,sqroot;
  for(counter=1;counter <=10; counter ++)
  { square= counter* counter;
    cout<<"\nnumber : "<<counter<< " square
: "<< square <<endl;
  }
}
/*Output: number : 1 : square : 1
number : 2 : square : 4
number : 3 : square : 9
number : 4 : square : 16
number : 5 : square : 25
number : 6 : square : 36
number : 7 : square : 49
number : 8 : square : 64
number : 9 : square : 81
number : 10 : square : 100*/
```

---

**Example 8.10: extern.cpp A  
Program to Demonstrate Usage of  
Extern Variable  
usage.usage.stack(auto) Variable**

---

```
Variable declared out side void main()
function comes under external storage
class.
#include<iostream>
using namespace std;
#include<conio.h>
// external function
void FindArea();
float x=10;float y=10;float area;
void main()
{ extern float area; extern float x;
extern float y;
    area=0;
    cout<<"\n area before calling
FindArea : "<<area<<endl;
    FindArea ();
    cout<<"\n area after calling FindArea
: "<<area<<endl;
    getch();
} // end of main
void FindArea()// fn declaration
{extern float x; extern float y;extern
float area;
    area = x*y;
    cout<<"\n area inside FindArea :
"<<area<<endl;}
/*Output:area before calling FindArea
0.000000
area inside FindArea 100.000000
```

```
    area after return from FindArea  
100.000000*/
```

---

Note that although FindArea did not return the value of the area, the main program has shown the correct value of area. Thus, we can use external declaration for avoiding passing of variable to and from functions. We can also use extern declarations and definitions to link variables declared in two different files. We will attempt to demonstrate the concept through our next example:

**Example 8.11: externfile.cpp A Program to Demonstrate Usage of External Program**  
**usage.usage.stack(auto) Variable**

External file is stored in another file. Write C++ code for void FindArea() and save it as findarea.h in the current directory.

---

```
#include<stdio.h>
void FindArea()
{ extern float x; extern float y;extern
float area;
    area = x*y;
    cout<<"\n area inside FindArea
:"<<area;
}
```

---

**Write C++ code for void main () and include it along with other include statements as shown below:**

---

```
#include<iostream>
using namespace std;
#include<conio.h>
#include"findarea.h" // findarea.h is
stored in the current working directory
void FindArea(); // external function
float x=10;float y=10;float area;
void main()
{ extern float area; extern float x;
extern float y;
    area=0;
    cout<<"\n area before calling FindArea
: "<<area<<endl;
    FindArea ();
    cout<<"\n area after calling FindArea :
"<<area<<endl;
```

```
    getch();  
} // end of main  
Note that you can use  
#include<findarea.h> if you copy the  
findarea.h  
into directory c:\tc\include/*  
Output: area before calling FindArea  
0.000000  
area inside FindArea 100.000000  
area after return from FindArea  
100.000000
```

---

## 8.11 Header Files and Standard Libraries

C++ language does not contain any built-in function. But it contains standard library provided by C++ supplier. The files stored in these library files, also called header files, contain all useful functions, their definitions, declaration of data types and constants. We have already used several of them like `iostream` and `conio.h`. A few of them are provided here:

---

**iostream.h** : contains basic stream IO files for c++  
**stdio.h** : facilitates input output statements.

**io manip.h** : Contains IO manipulators

**conio.h** : contains functions used in calling IO routines

**math.h** : Contains library functions.

Note that  $x$  can be float or double. C language does not support expression of type  $x^a$ . We need to use `pow(x,a)`.

`sqrt(x)` : determines square root of  $x$

`pow(x,a)` : computes  $x^a$

`exp(x)` : computes  $e^x$

`sin(x)` and `cos(x)` : computes sin and cosine

`log(x)` : computes natural log

`log10(x)` : computes log to base 10

**stdlib.h** : Contains standard library like search and sort routines, etc.

`abs(x)` : computes absolute value of integer  $x$

`atof(s)` : converts a string  $s$  to a double

`atoi(s)` : converts to integer

`malloc()` : allocates memory and returns a pointer to the location.

`calloc()` : same as `malloc()` but initializes the memory with 0s.

`free(x)` : frees heap memory space pointed by  $x$

`rand()` : generates a random number.

**string.h** : Contains string-related functions such as

`strlen()` : length of the char array

```
strcpy() : copies a string to
another
strcat() : concatenates two strings
strcmp() : compares two strings .
strlwr() : converts from uppercase
to lowercase
strupr() : converts from uppercase
to lowercase
strrev() : reverses a string
```

---

## 8.12 C++ Preprocessor

The preprocessor does some housekeeping function before submitting the source code to the C compiler. The jobs it performs are:

1. Inclusion of all include section files. For example, it fetches and appends `stdio.h` from `tc\include` directory and includes in source file.
2. Macro expansion. Actually preprocessor carries out substitution for declarations given in Macro statement. For example, in the statement  
`#define PI 3.14159`, preprocessor searches for occurrence of symbol `PI` and replaces it with `3.141519`.
3. Conditional inclusion. To prevent inclusion second and multiple number of times, we can employ conditional inclusion.
4. String replacements.

The standard directives available in C++ language are:

---

```
#include include text from the
specified file
```

```
#define define a macro
#undef undef a macro
#if test if a condition holds at
compile time
#endif end of if (conditional
preprocessor)
#elif if-else-if for multiple
paths at compilation time
#line provides line number
```

---

### *8.12.1 Macro Expansion*

#### **Example 8.12: macro1.cpp A Program to Show Macro Expansion. A Program to Show fn Overloading**

```
#include<iostream>
#include<conio.h>
using namespace std;
#define GETDATA cout << "Enter the
value: " <<endl
void main()
{int x , y;
GETDATA;
```



```

cin >> x ;
GETDATA;
cin >> y;
cout << endl << «values entered are «
<<x << « « << y;
getch();
}
// Output: Enter the value: 35
Enter the value: 45
values entered are 35 45*/
Preprocessor directive like #define
GETDATA cout << «Enter the value: «
<<endl will simply substitutes all the
occurrences of GETDATA with cout <<
«Enter the value: « <<endl

```

---

### *8.12.2 Macro Definition with Arguments*

The general syntax of macro with arguments is `#define macro-name (arg1, arg2, arg3,.....argn)`. Examples are shown below:

**Example 8.13: macro2.cpp. A Program to Demonstrate the Usage of Preprocessor Directives. A Program to Show fn Overloading**

```
#include<iostream>
#include<conio.h>
using namespace std;
// substitution & macro definition
macros
#define GETDATA cout << "Enter the
value: " <<endl
#define WRITEDATA(ans) cout << ans
#define SUM(a,b) ((a)+(b))
#define PRODUCT(a,b) ((a)*(b))
#define MIN(a,b) ((a)>(b)?(a): (b))
void main()
{ int x , y , ans;
  GETDATA;
  cin >> x;
  GETDATA;
  cin >> y;
  ans=PRODUCT(x,y);
  cout <<endl << «product of numbers is:
«;
  WRITEDATA(ans);
  ans=SUM(x,y);
  cout << endl << «sum of numbers is: «;
  WRITEDATA(ans);
  getch();
}
// OUTPUT: Enter the value: 7
Enter the value: 5
```

```
product of numbers is: 35
sum of numbers is: 12*/
```

---

### *8.12.3 File Inclusion*

All include files are placed at directory `c:\tc\include` for turbo C++ compiler or in `c:\program files\[Microsoft Visual Studio folder]\VC\include` for Microsoft VC++ compiler. Hence, if you copy any header file written by you into this directory, you can include this file as `#include <findarea.h>`. If you have placed the file in the current directory, you can include such a file with preprocessor directive `#include "findarea.h"`.

### *8.12.4 Conditional Inclusion*

Conditional Inclusion of some parts of source code is done using conditional inclusion macros. The syntax is

---

```
#if constant expression
    statement sequence
#endif
or
```

```
#if constant expression  
    statement sequence  
#else  
    statement sequence  
#endif
```

---

**Example 8.14: elseifmacro.cpp A Program to Demonstrate the Usage #if, #else and #Endif Preprocessor Directive. A Program to Show fn Overloading**

```
#include<iostream>  
#include<conio.h>  
using namespace std;  
#define UPPER 5000  
#define BONUS1 1000  
#define BONUS2 500  
#define bp 1000  
void main()  
{ int netpay;  
  cout << "The basic pay is: " << bp ;  
  #if bp < UPPER
```

```
netpay=bp + BONUS1;
#else
netpay=bp+BONUS2;
#endif
cout <<endl << "netpay = " << netpay;
getch();
} // end of main
//Output: Enter the basic pay 1000
netpay = 2000*/
```

---

### 8.12.5 Conditional Compilation *#ifdef* and *#ifndef* Statements

In order to prevent inclusion of macros more than once and leading to multiple declarations, we can use macro *#ifdef* and *#ifndef* to indicate if defined and if not defined. The general syntax is:

---

<i>#ifdef</i> macroname	<i>#ifndef</i>
Statements	statements
<b>or</b>	
<i>#endif</i>	<i>#endif</i>

---

### 8.12.6 *#undef*

A macro must be undefined before it is redefined. Consider the macro definitions.

In this module, we will also use conditional inclusion like `#ifdef`

### **Example 8.15: `undef.cpp` A Program to Show the Usage of Undefine**

```
#include<iostream>
#include<conio.h>
using namespace std;
#define UPPER 100
#define LOWER 10
#define WRITEDATA cout << "answer = " <<
ans << endl ;
void main()
{ int temp= 70,ans=10;
  if (temp<UPPER && temp>LOWER)
    WRITEDATA;
  #ifdef UPPER
  #undef UPPER
  #endif
  #ifdef LOWER
  #undef LOWER
  #endif
  // now we can set new limits for UPPER
```

```
& LOWER
#define UPPER 60
#define LOWER 50
if (temp<UPPER && temp > LOWER)
WRITEDATA;
getch();
} // end of main
//Output: Answer=10(printed from the 1st
if statement, the second WRITEDATA is
not executed as the 2nd if loop returns
a false.)*/
```

---

### 8.12.7 *#error* Macros

**#error** macro usage is shown below. When **#error** macro is encountered, the compiler displays the error message. Note that error message is not in double quotes

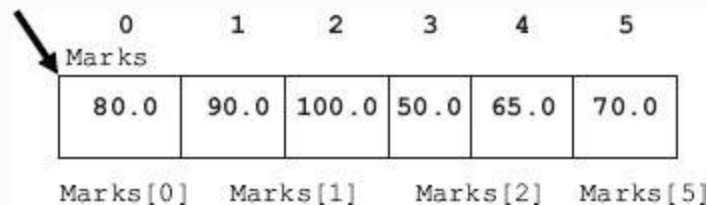
---

```
#ifdef UPPER
#include<upper.h>
#elifdef // this macro is similar to if-
else-if
#include<lower.h>
#else
#error Incorrect inclusion of Header
files
#endif
```

---

## 8.13 Arrays

In day-to-day life, there are several occasions wherein we have to store data of the same type in contiguous locations, like marks obtained by a student in six different subjects are shown in an array named ***marks*** in **Figure 8.2**. Elements of the array are referenced by array name followed by subscript. We have shown an array named marks; six subject marks scored by the student can be represented by `marks[0]=80.0`, `marks[5]=70.0`, etc.



**Figure 8.2** Representation of an array

**General syntax of array is: *storage class data type array [expression]***

Note that storage class is optional. Data type is data type of the array. Array is name



and expression is a positive integer.  
Examples of valid declarations of array are:

---

```
float marks[6] = { 60.0 , 66.0, 70.0 ,  
80.0 , 90.0, 100};  
float marks[] = { 60.0 , 66.0, 70.0 ,  
80.0 , 90.0, 100.0}; // no need to  
declare dimension  
char stg[]={ 'g' , 'o' , 'o' , 'd' };
```

---

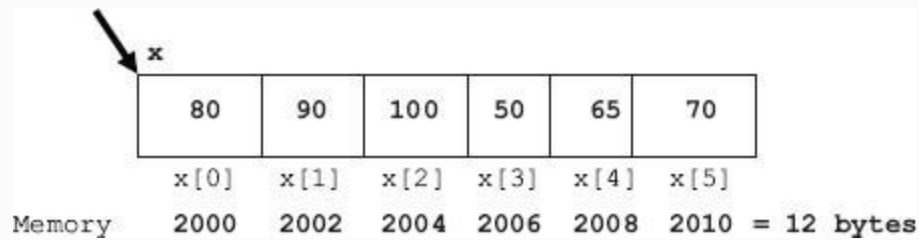
### *8.13.1 How are Arrays Stored in the Memory?*

Consider an array named x, declared as `int x[] = {80, 90, 100, 50, 65, 70};`

The addresses shown above are dummy addresses. Using of size of operators would tell us the memory requirement of data type `int` on your hardware. Assuming that it is 2 bytes, the memory of the array element is shown in Figure 8.2.

### *8.13.2 Array Initialization*

Refer to Figure 8.3 and the values stored in the array: `int x[6]`. We can initialize and allocate the values shown above with the statement



**Figure 8.3** Representation arrays with memory locations shown

```
int x[] = { 80,90,100,50,65,70;}
```

Alternatively, we can use cin and cout to obtain the values for array x as shown below:

```
cout<<"\n Enter Number of Terms in the  
array :";  
cin>>n;
```

**Example 8.16: array.cpp Write a Program to Display the Array Elements Along with their Address.**

## Sizeof Operator Provide Size of Data Type in Bytes.

```
#include<iostream>
using namespace std;
void main()
{ int i,n; //number of elements of array
x
  int x[]={80,90,100,50,65,70};
  n= sizeof(x)/sizeof(int);
  cout<<"\n size of data type <int>:"<<
sizeof(int)<<endl;
  cout<<"\n Memory space allocated to
x[6]:"<<sizeof(x)<<endl;
  cout<<"\n no of elements in array x =
"<<n<<endl;
  cout<<"\n array elements\taddress in
hexa ";
  for (i=0;i<n; i++)
  cout<<"\n"<<x[i]<<"\t\t"<<&x[i]<<endl;
  cout<<"\n";
}
//Output :size of data type <int>:4
Memory space allocated to x[6]:24
no of elements in array x = 6
array elements address in hexa
80  0012FF60
90  0012FF64
```

```
100 0012FF68
50   0012FF6
65   0012FF70
70   0012FF74*/
```

---

If your program exceeds at run time the space originally allocated when the array was declared, then array out of bounds error results. We can use `cin` to read into an array. In the following example, we will show use of **`cin.get()`** function when we reverse the string.

**Example 8.17: revstg.cpp//**  
**revstg.cpp A Program to Read the**  
**Input String Character by Character**  
**from Keyboard and Reverse the**  
**String**

---

```
// #include<iostream>
using namespace std;
// function prototype declarations
void main()
```

```

{ int count=0,i;
  int len; // length of the string
  char c;
  char x[20]; // array of characters .
string
  cout<<"\n Enter a word and press
<enter> :";
  c=cin.get(); // // get a character
  while ( c!='\n') // '\n' is end of
line character, i.e. pressing
  enter key
  {x[count]=c;
  count++;
  c=cin.get();
}
// we have reached end of line. Append
'\0' to the string
x[count]= '\0';
len = count;
// Now display the string you have just
read
cout<<"\n String inputted : "<<x;
cout<<"\n space allocated to single
char(byte) : "<<sizeof(char) <<endl;
cout<<"\n Memory space allocated to
string x[] : "<< sizeof(x) <<endl;
cout<<"\n No of characters in the string
x [] : "<<len<<endl;
// now reverse the string
cout<<"\n string X reversed : ";
for ( i=len-1;i>=0;i--)
  cout<<x[i];

```

```

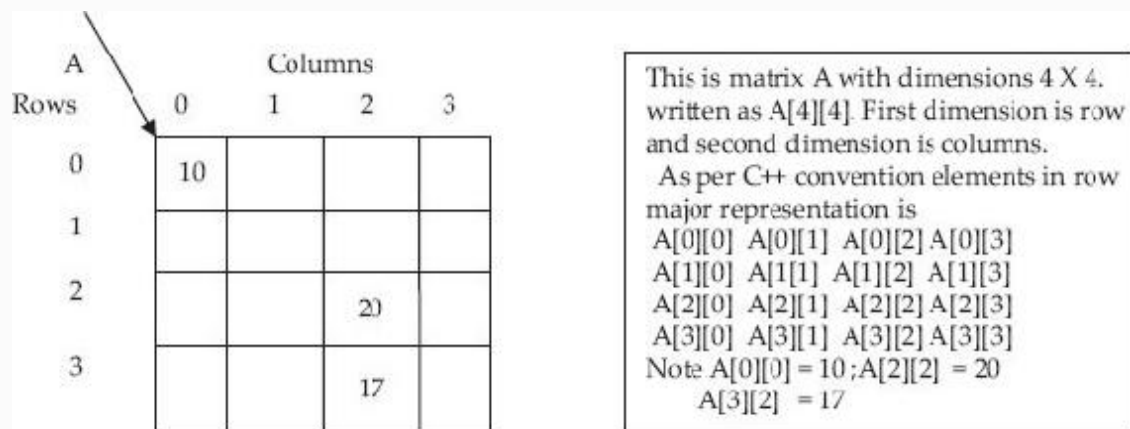
} //Output :Enter a word and press
<enter> :HELLO
String inputted : HELLO
space allocated to single char(byte) :1
Memory space allocated to string x[] :20
No of characters in the string x [] : 5
string X reversed : OLLEH */

```

---

### 8.13.3 Multi-dimensional Arrays

Arrays can have more than one-dimension. For example, a matrix is a two-dimensional array with a number of rows and a number of columns is shown in Figure 8.4.



**Figure 8.4** A two-dimensional matrix

## **Example 8.18: transpose.cpp. A Program to Find the Transpose of a Matrix**

```
#include<iostream>
using namespace std;
// functional prototype declarations
void Transpose( int A[10][10], int n
);// n is the order of square matrix
void ReadMatrix( int A[10][10], int n );
void PrintMatrix( int A[10][10], int n
);
void main()
{ int n,A[10][10];
  cout<<"Enter the order of square
matrix <n>";
  cin>>n;
  ReadMatrix(A,n);
  cout<<"The elements of the Matrix
are:"<<endl;
  PrintMatrix(A,n);
  cout<<"The elements of the Transpose
Matrix are:\n"<<endl;
  Transpose(A,n); //function call. A is
name of matrix. Name is address is
```

```

} //end of main
void Transpose(int A[10][10],int n)
//function definition*/
{ for(int i=0;i<n;i++) //loop1. i=1
because you don't have to touch
    x[0][0]
    { int t;
      for(int j=0;j<i;j++) //loop2
      {t=A[i][j];
       A[i][j]=A[j][i]; //swapping
       A[j][i]=t;
      } //end of loop2
    }
    PrintMatrix(A,n); // output the matrix
} //end of function transpose
void ReadMatrix( int A[10][10], int n)
{cout<<"Enter the elements :"<<endl;
  for(int i=0;i<n;i++)
    {for(int j=0;j<n;j++)
      cin>>A[i][j]; //input elements*/
    }
} //end of ReadMatrix
void PrintMatrix( int A[10][10], int n)
{for(int i=0;i<n;i++)
  {
    for(int j=0;j<n;j++)
    {
      cout<<"\t"<<A[i][j]; }
    cout<<"\n"<<endl;
  }
} //end of ReadMatrix
//output : Enter the order of square
matrix <n>2
Enter the elements

```



```
1 2 3 4
The elements of the Matrix are:
1 2
3 4
The elements of the Transpose Matrix
are:
1 3
2 4 */
```

---

### *8.13.4 Character Array – String Handling in C++ Language*

An array of characters is called a string variable. A string variable will always be automatically terminated with ‘\0’ (NULL) character. C++ compiler treats occurrence of NULL character to mean the end of string. In the following program, we would check for occurrence of ‘\0’ to indicate the end of strings. Consider string declaration shown below and decide which type of declaration is best for C++ language.

---

```
char city[6] ="Mumbai"; // incorrect as
no space for adding '\0' (NULL)
character.
char city[7] ="Mumbai"; // correct .
'\0' (NULL) character is added
```

```
automatically.  
char city[] = "Mumbai";    // correct .  
'\0' (NULL) character is added  
automatically.  
  
                                // This is the  
preferred mode and we will be using this  
mode.
```

---

## **Example 8.19: concat.cpp A Program to Concatenate Two Strings**

```
#include<iostream>  
using namespace std;  
// Fn Prototypes  
void Concat ( char x[],char y[]);  
void main()  
{ char x[20],y[20]; // x & y are two  
strings  
    cout<<"enter any 2 strings\n";  
    cin>>x>>y; //input 2 strings from the  
user*/  
    Concat(x,y); //function call*/  
} //end of main*/
```

```

void Concat(char a[],char b[])
//function definition*/
{ for(int i=0;a[i]!='\0';i++) // check
for '\0' occurrence
    cout<<a[i];
  cout<<» «;
  for(i=0;b[i]!='\0';i++)
    cout<<b[i];
} //end of function concat*/
// output :enter any 2 strings HELLO
BROTHER : HELLO BROTHER*/

```

---

The best way to learn programming is to write programs. Let us write our own code for achieving the above functionality. The problems and solutions are in the worked example section at the end of this chapter.

## 8.14 Summary

1. Functions help the main task to be decomposed into smaller modules. Calling function is one which calls a called function.
2. The function prototype is declared before, void main (), in the global area, so that it is accessible by all other functions.
3. Function takes care of all data types. One function executes for all data types depending on the arguments supplied at run time.

4. Mode of data transfer between calling function and called function, by copying variables listed as arguments into called function stack area and, subsequently, returning the value by called function to calling function by copying the result into stack area of the main function is called ***call by value***.
5. In call by reference, we pass the address to the function. Once the function receives a data item by reference, it acts on the data item and the changes made to the data item also reflect on the calling function.
6. Call by constant reference is used when we want to prevent values being changed by the receiving function.
7. Recursion means a function calling itself.
8. Inline functions mean C++ replaces the entire function code rather than resorts to the usual function-handling mechanisms.
9. Function overloading means one function performing more than one task depending on the arguments supplied at run time.
10. If values for the arguments function are not supplied, then function will assume default options. We must supply default arguments from left to right.
11. Variables declared as static are stored in contiguous memory locations. The only special feature of static area is that the variables in this static area are not erased, even if the function terminates.
12. After allocating mandatory spaces for all the above storage classes, the memory still free, is called heap memory or free space. But access to this space is available only through pointers.
13. Declarations and definitions using ***extern*** are available even to programs written separately with different file names but compiled and linked with the `main()` program.

14. The preprocessor does some housekeeping function before submitting the source code to the C compiler like inclusion of all include section files, macro expansion, conditional inclusion and string replacements.
15. Arrays store data of the same type in contiguous locations.
16. An array of characters is called a string variable. A string variable will always be automatically terminated with '\0' (NULL) character.
17. C++ has implemented all the functionality of C language string-based functions, compiler provides a full-fledged class called string. You can use `#include <string>` to realize the functionality.

## Exercise Questions

### Objective Questions

1. What is the value of `y = floor(35.5 )` ?

1. 35
2. 35.5
3. 36
4. 35.0

2. What is the value of `y = ceil(35.5 )` ?

1. 35
2. 35.5
3. 36
4. 35.0

3. `sizeof()` operator in C++ is a Library function TRUE/FALSE

4. `getch()` and `getche()` perform the same operation TRUE/FALSE

5. In `#include<iostream.h>` statement , `stdio.h`, under turbo C++, is available at

1. current directory
2. `tc\include`
3. `tc\bin`
4. none

6. In `#include "circle.h"` statement, `circle.h` is available at

1. current directory
2. `tc\include`
3. `tc\bin`
4. none

7. What will be output for `cout<< 65`?

1. a
2. A
3. 10
4. B

8. When two strings are equal `strcmp (stg1, stg2)` returns

1. -1
2. 0
3. 1
4. true

9. Call by reference is a default mode of transferring values to a function TRUE/FALSE

10. For recursive procedures, we can use the following storage allocations:

1. static
2. heap
3. stack
4. global

11. The following copies the value of an argument into the formal parameters of the subroutine:

1. call by value
2. call by reference
3. call by name
4. none

12. The following copies the value of an argument into the formal parameters of the subroutine:

1. call by value
2. call by reference
3. call by name
4. none

13. Character array must be terminated with

1. \0
2. \n
3. \a
4. \t

14. An array without initial values contains

1. all zeros
2. all 1s
3. garbage value
4. none of the above

15. Array can be initialized at the time of declaration itself using

1. [ ]
2. { }
3. ( and )
4. single quotes

16. The number in a square bracket of an array is called

1. superscript
2. subscript
3. dimension
4. range

17. Array declared as array  $A[m]$  the elements are subscripted between

1. 0...m

2.  $0 \dots m+1$
3.  $1 \dots m$
4.  $0 \dots m-1$

18. An array is always passed

1. using pass by value to a function
2. using pass by reference to a function
3. left to programs
4. pass by name

19. If `int A[6]` is a one-dimensional array of integers, which of the following refers to the value of the fourth element in the array:

1. `A[4]`
2. `A[2]`
3. `A[3]`
4. none

**Short-answer Questions**

20. Distinguish the switch and if–else statements.
21. When do you use for statement and while statements? State the situation when for statement is better than while statement.
22. Explain the differences of do-while and while statements.
23. Why is goto statement not preferred?
24. Explain the continue and break statements with examples.
25. Explain call by reference call by value.
26. Explain the need to declare function prototype before `main()` function.
27. What are the storage classes?
28. Explain the difference between static and stack storage.
29. Distinguish global and external declarations.
30. What is the role of preprocessors?
31. What are macros? Explain the `#ifdef` and `#ifndef` statements with examples.
32. What is a function template?



33. Distinguish call by value and call by reference.
34. What is call by constant reference?
35. What is function overloading?
36. What are preprocessor directives?
37. Why are inclusion directives important?
38. List a few important functions contained in String header file.

#### Long-answer Questions

39. Write in detail about one-dimensional and multi-dimensional arrays. Also write about how initial values can be specified for each type of array.
  1. In what way is array different from ordinary variable?
  2. What conditions must be satisfied by all the elements of any given array?
  3. What are subscripts? How are they written? What restrictions apply to the values that can be assigned to subscripts?
  4. What are the advantages in defining an array size in terms of a symbolic constant rather than a fixed integer quantity?
40. How are multi-dimensional **arrays defined**? Compare with the manner in which one-dimensional arrays are defined.
41. Explain memory mapping of C++ language.
42. What are storage classes? Explain with suitable examples.
43. What are macros and preprocessor directives? Explain with suitable examples.

#### Assignment Questions

44. Write a function template that returns position of a given value in an array.
45. Write a function template program to find sum of n natural numbers using recursion.
46. Write a macros in C++ for displaying error message using #error directive. Also program using macros to find the largest of the two numbers.

47. Write a function template to sort an array. Test your code for array of strings also by including string header in your program.
48. Write a function template to merge two sorted arrays.
49. Write a program to print the ASCII table for range 30 to 122.
50. Write a file named mystring.h, comprising all the above function modules at problem 12. Include the header file in your driver program and test all the modules.
51. Write a program to find the largest element in an array.

### **Solutions to Objective Questions**

1. a
2. c
3. True
4. False
5. b
6. a
7. b
8. b
9. False
10. c
11. a
12. b
13. a
14. c
15. b
16. b
17. d
18. b
19. c



# 9

## Pointers and References

### LEARNING OBJECTIVES

*At the end of the chapter, you should be able to understand and use*

- Pointers and references, and know when to use what?
- Objects and pointers.
- Objects and references.
- Dangling pointers and memory leaks.
- Constant pointers, constant objects and references.

### 9.1 Introduction

In our experience of teaching C++, we found that learners are pretty confused about

pointers and references and related memory management. This is mainly because they would have probably picked up the “what” of pointers and references and not the “why” of pointers and references. In this chapter, we would like to bring out special characteristics of pointers and references like

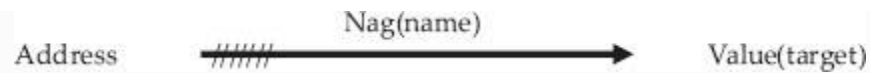
- Pointers are just like any other variables.
- Pointers have an address like a variable.
- Pointers point to a variable or a structure like array or an object.
- If you want a value of the object pointed by pointer just dereference the pointer.
- Operator new allocates memory for the pointer and delete frees the memory.
- The best part of pointers is that they can be reassigned.
- References are another name for the object.
- References always refer to the object.
- References cannot be reassigned.

In this chapter, starting from basics we would teach you how to handle pointers and references for C++ paradigm, without pitfalls such as dangling pointers and memory leaks.

## 9.2 What, Why and How of Pointers

In C++, we would like to use pointers because they point to location in memory and are very efficient to move multiple data items between the main program and function as function arguments. In addition, you can have pointers to a function and execute different functions just by reassigning the pointer based on the user's choice. Pointers would facilitate reassignment just like you can point at any person with your index finger.

How does Arjuna, the famous archer in Mahabharata, use his arrows? Firstly, he removed an arrow from his storage. Secondly, he gave a name (mantra like Nag Astra as shown in [Figure 9.1](#)). Then he points it toward the ground while thinking or getting address from his guide (Lord Krishna). He then pointed it to the ground so that the arrow did not take off accidentally and hit passers by or unintended targets. Lastly, he aimed at the target at the address given before letting it go! We will also do the same in the case of pointers.



**Figure 9.1** A pointer example with terms

---

```
int *ptr=0; // you have created a
pointer of type int
           // Note ptr is the pointer
           // Pointer is the address
           // *ptr is the value
           // you have now pointed to NULL
(0 is treated as NULL)
```

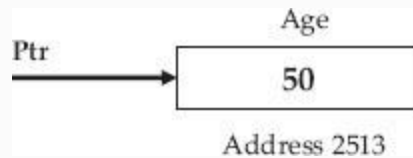
---

When `*` precedes a variable in declaration statement, it means that the variable is a pointer variable. For example, `ptr` in `int *ptr=0;` is a pointer variable. It is a good safe programming practice to initialize the pointer as soon as it is created.

### 9.3 Pointers Declaration and Usage

Let us say that at address 2513 we have stored an integer variable called `age` as

shown in Figure 9.2. At address 2613 we have integer variable age2 .



**Figure 9.2** Pointer in a memory

## Example 9.1: Pointers Declaration and Usage

```
int age=50;
int age2=18;
//we want pointer ptr to point to age;
ptr = & age; // you have assigned ptr
to age. & is called address operator
cout<< age; // displays 50
cout<<*ptr; // displays 50 . *ptr which
is value stored location, i.e. 50
//Once created you can reassign
pointers
```



```
ptr=& age2;
cout<<*ptr; // displays 18 . *ptr which
is value stored location, i.e. 18
```

---

**Dereference or indirection operator ( \*) is used to obtain the value pointed by the pointer. For example,**  
`cout<<*ptr; // displays 50`

### **Example 9.2: ptr1.cpp Pointers Usage**

```
#include<iostream>
using namespace std;
void main()
{ int age1=50;
  int age2=18;
  //create a pointer
  int * ptr=0;
  // assign it to age1
  ptr = & age1; // & is address of
operator
  cout<<"\n age1 "<< age1<<endl; //
```

```

displays 50
    cout<<"\n age1 with *ptr "<<
    *ptr<<endl; // displays 50 .*ptr is
value.
    cout<<"\n address using (&age1) and
ptr : "<<&age1<<"\t"<<ptr<<endl;
    // ptr is address of age1, so is
&age1. Hence both must be same
    // now we will reassign the same
pointer to age2
    ptr = & age2;
    cout<<"\n your age (*ptr) : "<<"\t"<<
    *ptr; // displays 18 .*ptr is value
stored location
    cout<<"\n address of (&age2) and ptr :
"<<& age2<<"\t"<<ptr<<endl;
    // ptr is address of age2 so is &age2.
Hence both must be same.
    cout<<"\n address of(&ptr) :"<<"\t"<<
    & ptr<<endl; // prints out address of
variable
    // ptr. We are not interested in this
address. It is just another address.
} //end of main
/*Output:
age1 50
age1 with *ptr 50
address using (&age1) and ptr : 0012FF7C
0012FF7C
your age (*ptr) : 18
address of (&age2) and ptr : 0012FF78

```

```
0012FF78
```

```
address of(&ptr) : 0012FF74
```

---

A note about address scheme of Intel processors is appropriate: When you ask for a display of address, the system would display a 32-bit address like `ffffffff4` in hexadecimal notation, which means

---

```
1111 1111 1111 1111 1111 1111 1111 0100  
in binary.
```

---

## 9.4 Call by Value and Call by Reference (Pointers)

What are they? You can pass variables to a function by either of:

1. **Call by Value:** The value of arguments is copied on to formal arguments whenever a function is called. Thus, there is an overhead of copying. As only copy reaches the function, the changes made in the local function are not reflected onto the original variable in the calling function. Further, if the data to be copied is large, ex. structure, the method is inefficient. It is hence used only for small data.
2. **Call by reference:** Actual arguments are not copied but only addresses (pointers) are forwarded. The functions get these addresses as arguments and works

on the variables located at the addresses forwarded. Hence, changes made to the variables are reflected on to variables from calling function. We are forwarding only addresses – there are no copying overheads as in call by value. Call by reference is of two types:

1. Call by reference using pointers
2. Call by reference using reference

Both types are effective. We will use pointers if reassigning is required, whereas if you use reference, its fixed memory location and reassignment to new location is NOT feasible.

### **Example 9.3: valptrref. Program to Highlight Call by Value and Call by Reference Using Pointers**

```
#include <iostream>
using namespace std;
// declaration of function prototypes
void Swap( int a , int b); // call by
value
void PtrSwap ( int *a , int * b); //
Call by Ref. a & b are pointers by def
void main()
```

```

    { int x=5;
      int y=10;
      // call by value
      Swap( x,y);
      cout<<"\nafter call by value :"<<endl;
      cout<<"x value after swap:"<<x<<endl;
      cout<<"y value after swap:"<<y<<endl;
      // call by reference using pointers.
Note that we have to send pointers
      //i.e. addresses of x & y . Hence we
will pass &x , and & y.
      PtrSwap(&x,&y);
      cout<<"\nafter call by ref using
pointers :"<<endl;
      cout<<"x value after Ptrswap:"
<<x<<endl;
      cout<<"y value after Ptrswap:"
<<y<<endl;
    } //end of main
// Function definitions
void Swap ( int a, int b)
{ int temp ; // two local variables
  temp=a;
  a=b;
  b=temp;
  cout<<"\ninside Swap using pointers
:"<<endl;
  cout<<"x value inside Swap:"
<<a<<endl;
  cout<<"y value inside Swap:"
<<b<<endl;
}

```

```

void PtrSwap ( int *a, int *b)
{ // a & b are pointers . Hence we
need a pointer called temp
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
    cout<<"\ninside PtrSwap using
pointers :"<<endl;
    cout<<"x value inside Ptrswap:"
<<*a<<endl;
    cout<<"y value inside Ptrswap:"
<<*b<<endl;
}
/*Output :inside Swap using pointers :
x value inside Swap:10
y value inside Swap:5
after call by value :
x value after swap:5
y value after swap:10
inside PtrSwap using pointers :
x value inside Ptrswap:10
y value inside Ptrswap:5
after call by ref using pointers :
x value after Ptrswap:10
y value after Ptrswap:5 */

```

---

**In Example 9.3, a few interesting results are given. Though inside Swap function values actually were interchanged, they were**

not reflected in the main program as discussed. Whereas in the case of `PtrSwap`, wherein we have passed pointers, the result of `PtrSwap` was reflected to main programs. This is in consonance with what we have learnt.

## 9.5 Dynamic Memory New and Delete Functions

We have learnt the memory management and mapping of C++ language in Chapter 3. In this section, we will learn tools and techniques to use heap memory, also called free space or dynamic memory, by pointers and references by using operators such as ***new and delete***.

Dynamic memory, or heap memory as it is known, affords very large programs to be run on limited primary memory resource. For example, the real memory requirement is known only at run-time from the user, whereas the memory allocation normally takes place at compile time. Further, to execute several of the overloaded functions, the system compiles, loads and then runs. So

if the user chooses at run time only one of the several functions loaded on to primary memory, the balance memory is wasted and hence we cannot solve problems requiring large memories. Dynamic memory solves this problem with new and delete operators. Allocate the heap memory with new operators and immediately after use, release the memory with delete operator. Thus, precious and limited memory is available for allocation and thus we can solve larger problems demanding larger memory. We can declare a heap variable using new operator.

---

```
int *x = new int; // creation and
allocation of heap memory
*x=25; // assign value
Alternately, we can use a single
statement
int *x = new int(12); allocate value 12
to pointer variable on heap memory.
```

---

Once allocated, you can use the dynamic memory pointer like an ordinary pointer. Remember that you have to release the memory after use so that the released



memory can be used for other heap memory requirements. In this way, we can solve more complex problems because compiler loads a heap variable and allocates the exact memory requirements indicated by the user as well as frees the memory after use. It is good practice to delete the dynamic memory allocated using delete operator. In fact, the number of new declarations must match the number of deletes, though the system automatically releases the memory once the program ends.

---

```
int *x = new int(12); // created a
pointer x on heap and allocated a value
=12
*q *q ; // find the square of value,
i.e. 144
cout<<*q;
.....
delete q; // q pointer has been deleted
```

---

### *9.5.1 Memory Leak*

If you reallocate the dynamic pointer to a new variable without deleting the existing assignment, the originally assigned variable and heap memory allocated is permanently

lost and not available to program. This is called memory leak.

### **Example 9.4: Memory Leak**

```
int *x = new int(12);  
*q *= *q ; // find the square of value  
i.e. 144  
    cout<<*q;  
int *x = new int(75); // Error since  
reassigned to new without deleting  
    delete q; // q pointer has been  
deleted.
```

#### *9.5.2 Dangling Pointer*

In this, the user tries to use the pointer after it has been deleted

### **Example 9.5: Dangling Pointer: An Example**

---

```
int *x = new int(12);
*q *= *q ; // find the square of value
i.e. 144
delete q; // q pointer has been
deleted.
cout<<*q; // Error. q has become
dangling pointer
```

---

### *9.5.3 Pointers and Arrays*

Let us understand the connection between pointers and arrays:

Note that 'x' is the name of the array. 'x' is also the address of the array and as well as the address of the first element of the array. Hence, we can call 'x' as the pointer to the array too.

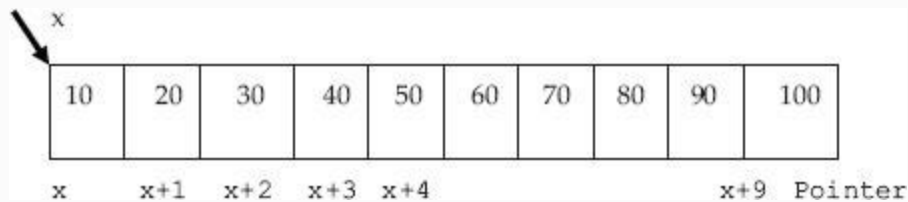
Suppose you want to print the 4th element, i.e. 40; as per C++ convention, you would write `cout<<x[3]`. Now as the element, we are interested in is at position 4 (3 in case we are counting from 0), i.e. at address `x+3`, we have learnt that if we want value from address we have to de-reference the address by using `*`.

---

```
Ex cout<< *(x+3);
```

---

Figure 9.3 shows array elements together with their addresses.



**Figure 9.3** Array with addresses (pointers)

#### 9.5.4 Pointers and Two-dimensional Arrays

Let us say that you have a two-dimensional array 'a' with 12 rows and 20 columns.

Then we can declare it as:

```
int a[12][20]  
or
```

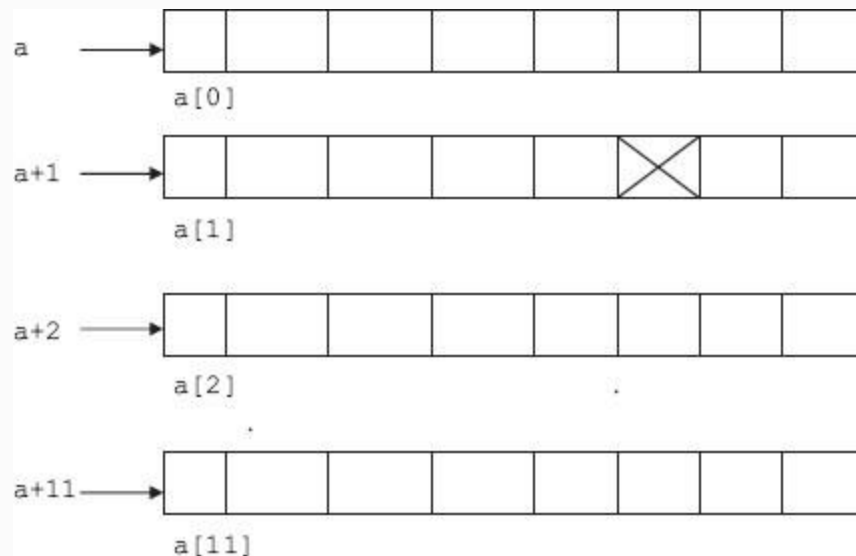
as a one-dimensional array using pointers.  
For example:

```
int *a[20]
```

---

We can also depict the above  $a$  as *pointer to pointer\*\* a*

Pictorially following the figure makes the concept clear. Therefore,  $a[0]$  points to the beginning of the first row.  $a[11]$  points to the beginning of the 11th row. Note that  $a[0] \dots a[11]$  are pointers to respective rows.



Now suppose you want to access the 1st row 5 element; then  $a[1]$  is the pointer to the first row and 5 elements displacement is 5:

We know we can write  $a[1]$  as  $*(a+1)$

Therefore, the address of the desired element is  $a[1] + 5$  or  $*(a+1) + 5$

The value of element is :  $*(*(a+1) + 5)$  .

### *9.5.5 Array Declaration on Heap Memory*

We can declare an array of 12 integers as :  
`int x[12]` . This array declaration reserves 12 contiguous locations in memory. In case the user does not use all 12 locations, memory will be wasted. Array declaration is an example of static memory allocation. Instead, we can also declare an array as

---

```
int *x =new int[12]; // x is a pointer
variable pointing to array by name x
//, having 12 contiguous locations.
int *ptr=x; // now ptr points x[0], i.e.
starting of an array
```

---

Let us write a program to pass an array to a function that receives an array by reference and sorts an integer array. We can define an array of 12 integers using new function and release the heap memory after use by deleting function, as follows:

## **Example 9.6:    dynarray.cpp A Program for Sorting of an Array by Passing an Array by Reference and to Find the Maximum of an Array**

```
#include<iostream>
using namespace std;
// function prototypes
template<class T>
T FindMax( T x[] , int n);
template<class T>
void SortArray( T *a , int n);
void main()
{ int n;
  int max; // n=no of values in an array
  int *x; // x is a pointer to an array
  cout<<"how many elements in your array
:";
  cin>>n;
  // allocate dynamic memory space using
new operator
  x=new int[n]; // allocates 12
contiguous location to pointer x
  // read in the array
  for (int i=0;i<n;i++)
```

```

        { cout<<"value for "
<<i+1<<"element:";
            cin>>x[i];
        }
    cout<<("\n The entered array is...\n");
    for (i=0;i<n;i++)
        cout<<" "<<* (x+i); // same as writing
x[i]
// call Findmax function
    max=FindMax(x,n); //x is a pointer to
array
    cout<<"\n maximum value of given
array="<<max<<endl;
// call SortArray function
    SortArray(x,n);
    cout<<("\n The entered array is...\n");
    for (i=0;i<n;i++)
        cout<<" "<<x[i]; // same as writing *
(x+i)
    // now that our work with array is over
let us delete the memory allocated
    delete [] x; // memory allocated with
new operator stands released
} //end of main
// Fn definition
template<class T>
T FindMax(T x[], int n)
{ T max;
    int i;
    max=T(); // *x is the value of 1
element
    for(i=1;i<n;i++)

```



```

        { if (max<*(x+i))
          max=*(x+i);
        }
    return max;
} // end of FindMax
template<class T>
void SortArray( T *a ,int n)
    { int i,j; // i for outer loop j for
inner loop and temp for swapping
    T temp=T();
    for ( i=0; i< n-1; i++) // last value
need not be sorted
        {
            // find the smallest of
remaining numbers and exchange it with
            for ( j=i+1; j< n; j++)
                { if (*(a+j) < *(a+i))
                    { // swap
                        temp=*(a+i);
                        *(a+i)=*(a+j);
                        *(a+j)=temp;
                    }
                }
        }
    }
}

/*output : how many elements in your
array :6
value for 1element:32
value for 2element:43
value for 3element:34
value for 4element:65
value for 5element:67

```

```
value for 6element:12
The entered array is...
32 43 34 65 67 12
maximum value of given array=67
The sorted array is...
12 32 34 43 65 67
```

---

### *9.5.6 Pointer to Pointer*

You remember the treasure hunt game we all have played at some time or the other. In a first clue, we would receive a chit that gives clues regarding a second address at which the second clue or treasure is kept. Pointer to pointer can be viewed as pointer to an address, i.e. an address that points to another address. We can recover the value from the second address. For example, consider a two-dimensional matrix.

Let us say that you have a two-dimensional array 'a' with 12 rows and 20 columns.

Then we can declare it as:

---

```
int a[12][20]
or
```

---

as a one-dimensional array using pointers.  
For example:

---

```
int *a[20]
```

---

We have learnt in Section 9.4.4 that we can also depict the above **a** as pointer to pointer **\*\* a**. We have shown deployment of dynamic memory allocation techniques in Example 9.4.

When will the pointer to pointer be useful? An array or function is known by its name. We have also learnt by now that name is address. Symbol Address Table stores names and also address allocation for functions, arrays, variables, etc. Now in situations wherein there is a need to delete a first entry like the first element of the array or there is a need to add an element in the front, a normal pointer would entail changing of name in the symbol address table and entail in fructuous work. A pointer to pointer would isolate names and addresses stored by the system and hence adding in the front and deletion would become less cumbersome.

### *9.5.7 Dynamic Memory for a Two-dimensional Array*

In our next example, we will demonstrate the creation and usage of dynamic memory using new operators. We will read a  $2 \times 2$  matrix. First, we will allocate dynamic memory for rows and then we will allocate for columns of each row.

We will use try and catch blocks. This feature will be explained in detail in the chapter on errors and exceptions, but for now, we will introduce the concept in brief here. When an error occurs during the execution of a program, due to non-availability of resources, normally the program stops execution and reports the error or exception. Try and catch blocks provide a way out for the programmer to take corrective steps for the errors and exceptions that occur during run time. Try block will allocate heap memory during run time. If it fails due to nonavailability of memory or for any other reason, then it will throw the exception object. Catch block will

catch the exception thrown by try block and take remedial measures.

### **Example 9.7: matptrptr.cpp A Program for Reading and Printing of Matrix with Dynamic Memory Allocation Using Pointer to Pointer**

```
#include<iostream>
using namespace std;
// functional prototype declarations
void ReadMatrix( int **A,int m , int n);
void PrintMatrix( int **A , int m , int
n);
void main()
{ int m,n;
  int **A=0; // A is a pointer to
pointer
  cout<<"Enter the order of matrix <m,n>
:";
  cin>>m>>n;
  /* Now allocate dyn memory. First
allocate to rows
  Then allocate to columns. We will use
try and catch blocks.
  Try will allocate dyn memory. If it
```

fails due to non-availability of memory then it will throw the exception object.

Catch block will catch the exception thrown by try block and take remedial measures. \*/

**try**

```
{    A=new int * [m]; // dynamic memory
for row allocation
    for (int i=0;i<m;i++)
        A[i]=new int[n]; // dynamic memory
for columns
}
```

**catch** (bad\_alloc)

```
{            cout<<"\n bad
allocation"<<endl;
            exit(1);
}
```

```
    ReadMatrix(A,m,n);
    printf("The elements of the Matrix
are:\n");
```

```
    PrintMatrix(A,m,n);
}    /*end of main*/
```

void ReadMatrix( int \*\*A, int m,int n)

```
{    int i,j;
    cout<<"Enter the elements :";
    for(i=0;i<m;i++)
        { for(j=0;j<n;j++)
            cin>>A[i][j]; /*input
elements*/
    }
```

}//end of ReadMatrix

void PrintMatrix( int \*\*A, int m,int n)

```

{    int i,j;
  for(i=0;i<m;i++)
  {   for(j=0;j<n;j++)
      cout<<" "<<A[i][j];
      cout<<endl;
  }
}
/*output: Enter the order of matrix
<m,n> :2 2
Enter the elements :1 2 3 4
The elements of the Matrix are:
1 2
3 4*/

```

---

### *9.5.8 Pointers and Three-dimensional Arrays*

To access an element  $a[3][4][5]$

1.  $a$  is the pointer to the first row. We need three rows. Therefore, it is  $a[3]$  or  $*(a+3)$ .
2. Column displacement is 4. Therefore, the address is  $*(a+3)+4$  and the value is:  $*(*(a+3)+4)$ .
3. Three-dimensional displacement is 5. Therefore, the address is:  $*(*(a+3)+4)$ . The value of element, therefore, is:  $*(*(*(a+3)+4))$ .

### *9.5.9 Array of Pointers*

Pointers can be stored in arrays. You already know that pointer means address, hence an array of pointers means a collection of addresses. For example, you can store a set of five pointers each pointing to a string variable like:

---

```
char * ptr[5] = { "welcome", "to"  
,"self_learning","C++" , "Book" }
```

---

ptr is a dimension 5, i.e. an array of five pointers. The following example will make the concept clear:

### **Example 9.8: arrayofptr.cpp**

#### **Program to Demonstrate Use of Array of Pointers**

```
#include<iostream>  
using namespace std;  
void main()  
{ // array of pointers with 5 elements  
char *ptr[5]=  
{"welcome","to","self_learning","C++A","
```



```

Books"};
char *x; // x is a pointer of type char
x=ptr[0]; // x now points to ptr, i.e.
starting pointer in an array of pointers
for ( int i = 0 ; i< 5; i++)
    cout<<"\n"<< *(ptr+i)<<endl;
// following cout statements will teach
you more about array of pointers
    cout<<"\n"<< *ptr[3]; // you can expect
value of starting element in CDS i.e. C
}
/*Output:
welcome
to
self_learning
C++
Books*/

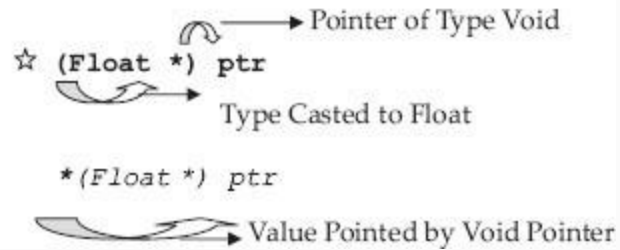
```

---

### *9.5.10 Pointers to Void*

Remember 'void' is a data type. Usually, we will declare pointer to point to a particular type of data. For example, `int * ptr` or `char * ptr`, etc. The void can be employed if your program has multiple data types. But typecasting is essential, when the void pointer is used as shown in [Figure 9.4](#).

### Type Casting a Void Pointer



**Figure 9.4** Void pointer declaration, definition and reassignment

## Declaration:

```
int x;  
float salary;  
void *ptr ; // pointer to void data type  
declaration
```

## Definition:

```
ptr = & x ; // assign pointer to type  
integer
```

## Usage:

Type casting the void pointer is mandatory

```
cout<<* (int*)ptr;
```

## Reassignment:

---

```
ptr=&sal;  
cout<< * (float*)ptr; // typecasting of  
ptr to float
```

---

### **Example 9.9: voidptr.cpp A Program to Demonstrate the Use of Void Pointers**

```
#include<iostream>  
using namespace std;  
#include<iostream>  
using namespace std;  
void main()  
{ int x=100;  
  float sal=2000.00;  
  void * ptr;// ptr is a pointer to  
data type void  
  // assign void pointer to int  
  ptr=&x;  
  cout<<"\nptr type casted to integer  
:"<< *(int*)ptr; // typecasting of ptr  
to int
```

```
// assign void pointer to float
ptr=&sal;
cout<<"\nptr type casted to float
:"<<*(float*)ptr; // typecasting of ptr
to float
} //end of program
/*Output : ptr type casted to integer
:100
ptr type casted to float :2000*/
```

---

### *9.5.11 Pointer to a Constant vs const Pointer*

We are aware that pointers are addresses and provide us a fast access to memory and data manipulation. But there are issues of safe operation and data integrity. There are situations wherein we want that data should not be altered. In such cases, we would have declared that data as constant. The main advantage of pointers is that they can be reassigned. This feature allows us to navigate through the array or memory, etc. But there are situations when we need the pointers to be constant, i.e. they should not be reassigned. In such cases, we would declare const pointers.

Thus, there are two combinations involved, *i.e. **pointer type and data type***. Each type can have two variations of data type: constant and normal. Therefore, we can have a total of four variations in the declaration of variables. These are shown below:

For example, let us say we have two variables declared as

---

```
int val = 25;  
const float PI=3.14159;
```

---

## **Pointer to data // normal pointer and normal data**

---

```
int *x=& val;  
Pointer to const data  
// y is a pointer to constant float. It  
cannot be changed const float * y=&PI;  
Const pointer to data  
// constant pointer to int data type.  
Pointer cannot be reassigned int * const  
x=& val;  
Const pointer to a const data  
// constant pointer to constant data  
type. Hence value cannot be changed  
// and pointer cannot be reassigned  
const float * const z=& PI
```

---

## **Example 9.10: Pointers Constant Data and Constant Pointers. Constructors and Destructors of a Class**

---

```
// pointer to int
int val = 25;
// int variable
int *ptr = & val ;           // ptr
is a normal pointer
cout<<++(*ptr);      // output 26
cout<<++ptr;         // ptr is incremented
// constant pointer to int
int const * cptr = & val; // cptr is
constant pointer to int
cout<<++(*cptr);     // Allowed . output
26
cout<<++cptr;        // Not allowed . cptr
cannot be incremented
// pointer to constant float PI
const float PI=3.14159;
// float variable PI declared as
constant
```

```

const float * cfptr = &PI;           //
cfptr is pointer to const data type
float
cout<<++(*cfptr); // Not Allowed as
*cfptr is constant
cout<<++cfptr;      // Allowed .
Increments cfptr
// constant pointer to constant float PI
const float * const cfptrc = &PI ; //
cfptr is const pointer to const data
cout<<++(*cfptr);           // Not
Allowed as *cfptr is constant
cout<<++cfptr;              // Not
Allowed . cfptr is constant.

```

---

### 9.5.12 Pointers to Function

We are aware a name means an address. Like array, name is address of the array. Hence, we can call it as pointer to an array. Function name likewise is an address. We can think of it as a pointer to a constant. At that address, the code for the function is stored.

Pointer to function hence means pointer to a constant pointer.

---

```

int Findmax( int a[] , int n); // A
function is declared

```

```
int (*ptrfun ) (int) ; // a pointer to
function i.e. ptrfun is created
ptrfun = & FindMax ; // pointer to
function is assigned to FindMax function
```

---

Note that in the statement `int (*ptrfun ) (int) ;` the function takes `int` as argument and returns `int` data type.

### **Example 9.11: ptrtofun.cpp. Program to Demonstrate Use of Pointers to Function**

---

```
#include<iostream>
using namespace std;
void FindTriangle(float &a , float & b
);
void FindRect(float &a , float & b );
void (*funptr) ( float & , float & ); //
ptr to function
void main()
{ float x= 25.0 , y=50.0;
  int choice;
  cout<<"\n Enter your choice .. < 0 to
```



```

quit : 1 for Triangle : 2 for Rectangle
>";
    cin>>choice;
    while ( choice !=0)
    {        switch (choice)
        case 0 : cout<<"\n exiting the
programme..."<<endl;
                exit(0);
        case 1 : funptr = FindTriangle;
break;
        case 2 : funptr=FindRect; break;
    }// end of switch
    funptr(x,y);
    cout<<"\n Enter your choice .. < 0 to
quit : 1 for Triangle : 2 for Rectangle
>";
    cin>>choice;
} //end of while
} //end of main
//Fn definitions
void FindTriangle( float &a , float &b)
{cout<<"\n Area of the Triangle = "<<
0.5*a*b << endl;}
void FindRect( float &a , float &b)
{cout<<"\n Area of the Rectangle = "<<
a*b << endl;}
/*Output : Enter your choice .. < 0 to
quit : 1 for Triangle : 2 for Rectangle
>1
Area of the Triangle = 625
Enter your choice .. < 0 to quit : 1 for
Triangle : 2 for Rectangle >2

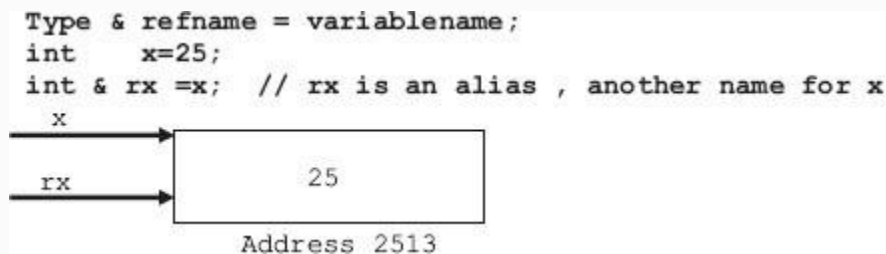
```

```
Area of the Rectangle = 1250
Enter your choice .. < 0 to quit : 1 for
Triangle : 2 for Rectangle >0*/
```

---

## 9.6 What, Why and How of References

C++ gives us an additional facility called reference. A reference is an alias or another name to a variable. Figure 9.5 shows usage of reference.



**Figure 9.5** Reference `x` and `rx` refer to the same location 2513 and value 25

### Reference operator `&`: It has two uses

---

```
&x ; // evaluates the address of x
int & rx = x ; implies that rx is a
reference of type int
```

---

## Example 9.12: ptrtofun.cpp. 6 ref1.cpp Program to Introduce Reference Concepts

```
#include<iostream>
using namespace std;
void main()
{   int age1=50;
    int age2=18;
    //create a reference to age1 i.e. an
    alias by name rage to age1
    int & rage=age1; //rage is an alias or
    another name to age1
    cout<<"\n age1 "<< age1<<"\t"<<"&age1
    "<< &age1<<endl; // displays address of
    age1
    cout<<"\n age1 with rage"<<
    rage<<"\t"<<"&rage "<< &rage<<endl;
    cout<<"\n both rage and age1 must have
    same addresses.."<<endl;
    /* you cannot reassign references like
    you did for pointers. If you do, the
    original assigned data will be lost*/
    rage = age2;
```

```

cout<<"\n wehave forcibly reassigned a
reference to ";
cout<<"\n another variable. Rage is an
alias of age1.";
cout<<"\n Therefore age1 will now have
value of age2."<<endl;
cout<<"\n age1 "<< age1<<"\t"<<"&age1
"<< &age1<<endl;
cout<<"\n age2 "<< age2<<"\t"<<"&age2
"<< &age2<<endl;
cout<<"\n rage assigned to age2 but
value it hold is age1 :"<<rage<<endl;
} //end of main
/*Output : age1 50 &age1 0012FF7C
age1 with rage50 &rage 0012FF7C
both rage and age1 must have same
addresses..
we have forcibly reassigned a reference
to
another variable. Rage is an alias of
age1.
Therefore age1 will now have value of
age2.
age1 18 &age1 0012FF7C
age2 18 &age2 0012FF78
rage assigned to age2 but value it holds
is age1 :18*/

```

---

### ***9.6.1 Which is Better – Pointer or Reference?***

**Pointers** are used when you have to reassign. For example, if you are using an array that stores values at contiguous memory locations, you will need to increment the pointer to refer to the next element of the array. But pointers, though quick acting, have inherent dangers associated with them like memory leak, memory crash, dangling pointers, etc.

**References** are aliases – another name for variables. They cannot be reassigned. If you insist and reassign, as reference is an alias, the first variable will be assigned with reassigned variables value.

**Common advantages of both pointers and references.** Both forward only addresses to function, i.e. they employ pass by reference technique. Hence, there are no overhead like making copies of variables and objects while passing these items to a function. Function can return more than one value.

References offer a clean efficient and risk and error-free environment and hence it is the experienced programmers first choice. We will present the advanced concepts

involved with references through a series of programs in our chapter on objects. References offer risk-free and efficient programming. To start with, recall how PtrSwap program in Example 9.2 ensured that changes made in function reflected in the main program. In our next example, we have shown how we can achieve the same result by using **pass by reference**.

### **Example 9.13: RefSwap.cpp. A Program to Swap Values by Using References Concepts**

```
#include<iostream>
using namespace std;
// declaration of function prototypes
void RefSwap ( int &a , int &b); //
Call by Ref. a & b are pointers by def.
void main()
{   int x=5, y=10;
    // call by reference using reference.
    Note that we have to send reference
    //In prototype we have promised to
```

```

send references like &a and &b
    //i.e. addresses of x & y . Hence we
will pass x and y.
    RefSwap(x,y);
    cout<<"\nafter call by ref using
References :"<<endl;
    cout<<"x value after
RefSwap:"<<x<<endl;
    cout<<"y value after
RefSwap:"<<y<<endl;
} //end of main
// Function definitions
void RefSwap ( int &a, int &b)
{ // a & b are references . Hence we
need ordinary variable temp
    int temp; temp=a; a=b; b=temp;
    cout<<"\ninside RefSwap using
reference :"<<endl;
    cout<<"x value inside
RefSwap:"<<a<<endl;
    cout<<"y value inside
RefSwap:"<<b<<endl;
}
/*Output:
inside RefSwap using reference :
x value inside RefSwap:10
y value inside RefSwap:5
after call by ref using References :
x value after RefSwap:10
y value after RefSwap:5 */

```

---

## Example 9.14: RefSwap.cpp. A Program to Swap Values by Using References Concepts

In this example, we will show how to return a reference from a function `FindLarge`. We declare a function prototype as `double & FindLarge(double &r, double &s)`.

The function finds the largest of the two double quantity and returns a reference to largest, in this case to `m`. Also observe the working of **`FindLarge (k, m)=10 ; & FindLarge (k, m) ++;`** statements.

```
#include <iostream>
using namespace std;
double & FindLarge(double &r, double
&s);
int main ()
{ double k=5, m=9;
  cout <<" Given Values k: " << k <<" :
m: " << m << endl;
  cout <<" \nValues after FindLarge (k,
m) : "<<FindLarge (k, m)<<endl;
```



```

        cout <<" Values k: " << k <<" : m: "
<< m << endl; //output 9
        cout << endl;
        FindLarge (k, m)=10; // largest is m=9
, it is replaced by 10 .
        cout <<" \nValues after FindLarge (k,
m) = 10"<<endl;
        cout <<" Values k: " << k <<" : m: "
<< m << endl; //output 3 & 10
        cout << endl;
        FindLarge (k, m) ++;
        cout <<" \nValues after FindLarge (k,
m)++"<<endl;
        cout <<" Values k: " << k <<" : m: "
<< m << endl; //output 3 & 11
        cout << endl;
        return 0;
}
double & FindLarge(double &r, double &s)
{if (r > s) return r;
    else return s;
}
/*output :Given Values k: 5 : m: 9
Values after FindLarge (k, m): 9
Values k: 5 : m: 9
Values after FindLarge (k, m)=10
Values k: 5 : m: 10
Values after FindLarge (k, m)++
Values k: 5 : m: 11 :*/

```

---

## 9.7 Summary

1. Pointers are like variables. Hence, they have address.
2. Pointers are addresses.
3. To get a value, use dereference operator \* on pointer.
4. Pointers can be reassigned to point to any other variable of the same data type.
5. Operator new allocates space for pointer and statement delete frees the dynamic pointer.
6. Memory leak is reallocating the dynamic pointer to a new variable without deleting the existing assignment; the originally assigned variable and heap memory allocated is permanently lost and not available to program.
7. Dangling pointer: In this, the user tries to use the pointer after it has been deleted.
8. Array of pointers. Pointers can be stored in arrays. Array of pointers means collection of addresses.
9. Void pointers point to data type called void. Explicit type casting is required when void pointers point to a particular type of data.
10. Pointer to function. It is as a pointer to a constant pointer as function name is an address. At that address, the code for the function is stored.
11. A reference is an alias or another name to a variable. A reference always points to a variable.
12. A reference cannot be reassigned. If reassigned, the original variable will lose its data.
13. A reference is risk free and free from memory leaks and dangling pointers.
14. A reference is best suited when reassignments are not present or when we need fixed address.
15. A pointer, on other hand, is most suited when reassignment is required.

# Exercise Questions

## Objective Questions

1. The pointer is used to specify a pointer whose base type is unknown and is a generic pointer.
  1. NULL pointer
  2. NIL pointer
  3. 0
  4. Void
2. \_\_\_\_\_ is the means by which a program can obtain memory during run time.
  1. Static allocation
  2. Dynamic allocation
  3. Stack allocation
  4. All of a, b, c.
3. Memory allocated by C++'s dynamic allocation functions is obtained from \_\_\_\_\_ memory.
  1. Global
  2. Static
  3. Stack
  4. Heap
4. The new function returns a pointer of type \_\_\_\_\_ which means that we can assign it to any type of pointer.
  1. NULL pointer
  2. Void
  3. 0
  4. NIL
5. In call by reference, actual arguments are not copied but only addresses (pointers) are forwarded. True/False
6. The declaration of two-dimensional arrays `int a[12][20]` and `int *a[20]` in dynamic memory allocation scheme are one and the same. True/False

7. Multiplication and division operations are allowed on pointers. True/False
8. Only addition and subtraction operations are permitted on pointers. True/False
9. The \_\_\_\_\_ is a unary operator that returns the memory address of its operand.

1. &
2. +
3. %
4. \*

10. Which among these is the indirection operator?

1. &
2. +
3. %
4. \*

11. Dereferenced means get the value stored at location. True/False
12. Indirection operator and dereferencing mean one and the same. True/False
13. Const pointer can call a non-constant function. True/False
14. If object is declared constant, then this pointer is a constant pointer. True/False
15. Dynamic memory can be allocated without the use of pointers. True/False
16. If `int A[6]` is a one-dimensional array of integers, which of the following refers to the value of fourth element in the array:

1. \* (A+4)
2. \* (A+3)
3. A+4
4. A+3

17. `cout<< x[5];` is equal to

1. `cout<< *x[5];`

```
2. cout<< x+5;  
3. cout<< *x+5;  
4. cout<< *(x+5);
```

18. Constant this pointer can call non-constant functions. True/False
19. Pointer to a function is pointer to a constant pointer True/False
20. If ( a= =b) then ( &a = = & b) True/False
21. If ( a= =b) then ( \*a= =\*b) True/False

#### Short-answer Questions

22. What are pointers? List out reasons for using pointers.
23. Explain the process of assessing a variable through its pointer. Give an example.
24. How to use pointers as arguments in a function? Explain through an example.
25. Explain the process of declaring and initializing pointers. Give an example.
26. Distinguish pointer \* operator (indirection operator) and address operator(&) with examples.
27. Explain passing by reference using pointers and using reference.
28. Which usage is better: pointers or reference? Why?
29. Distinguish & operator and reference in C++ language.
30. Differentiate array of pointers and pointer to an array.
31. Explain heap memory space or free storage space.
32. Explain dynamic memory operators new and delete.
33. How do you declare an array on heap memory? Explain with examples.
34. How do you allocate dynamic memory for a two-dimensional array using pointer to pointer concept.
35. Explain how a void pointer could be useful.
36. Explain pointer to a function with examples.
37. Distinguish dangling pointer and memory leak with examples.

38. Explain different ways to use & operator.
39. Distinguish reference operator and dereference operator.
40. Explain pointer to pointer declaration in case of two-dimensional array.

#### **Long-answer Questions**

41. Write a cpp that forwards an array of strings of names function that return a new array that contains pointers to duplicate names.
42. Write a cpp that reverses an array of integers.
43. Write a cpp to find the trace of a square matrix. [Hint: Trace is sum of diagonal elements.]
44. Write a cpp to find if the given matrix is singular matrix. [Hint: Matrix is singular if determinant is 0. Use recursion to find the determinant of a matrix.]
45. Explain how dynamic memory can be allocated to two-dimensional array when the dimensions are not known at compile time. [Hint: Use pointer to pointer concept.]
46. The declarations `int a[5][6]` and `*a [6]` and `**a` all refer to the same two-dimensional array of integers. Explain the process of memory allocation for the above three declarations.
47. Write a cpp to implement Merge Sort.

#### **Solutions to Objective Questions**

1. d
2. b
3. d
4. b
5. True
6. True
7. False

- 8. True
- 9. a
- 10. d
- 11. True
- 12. True
- 13. False
- 14. True
- 15. False
- 16. a
- 17. d
- 18. False
- 19. True
- 20. True
- 21. False

# 10

## Classes

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to*

- Write C++ programs using classes and objects.
- Understand concepts of constructors, destructors, and member data and functions.
- Understand container classes and their usage.
- Understand friend functions and inline functions.
- Understand objects and dynamic memory usage with pointers and reference.
- Understand *this* operator and static declarations and their usage.

### 10.1 Introduction



We use class in C++ to define our own data type. A class is a derived data type like an array. The difference is that in an array you have single data type, while in a class data type you can have different data types. These different data types can be **intrinsic data types** such as int, float, etc. or **derived or user-defined data types** or *functions or operators*.

**Object** contains data types and functions and/or operators defined by class. So we can call **object as an instance** of a class. In other words, **object is a variable of data type class**. We can define an object as an independent entity that holds its own data and member functions. We can say that object tells us the data it holds and allowable operations on the object. **A class therefore allows us to encapsulate member functions and member data into single entity called object.**

Object-oriented programming involves writing programs that use classes. We create objects of different classes to solve the problem at hand and make objects communicate with each other through the

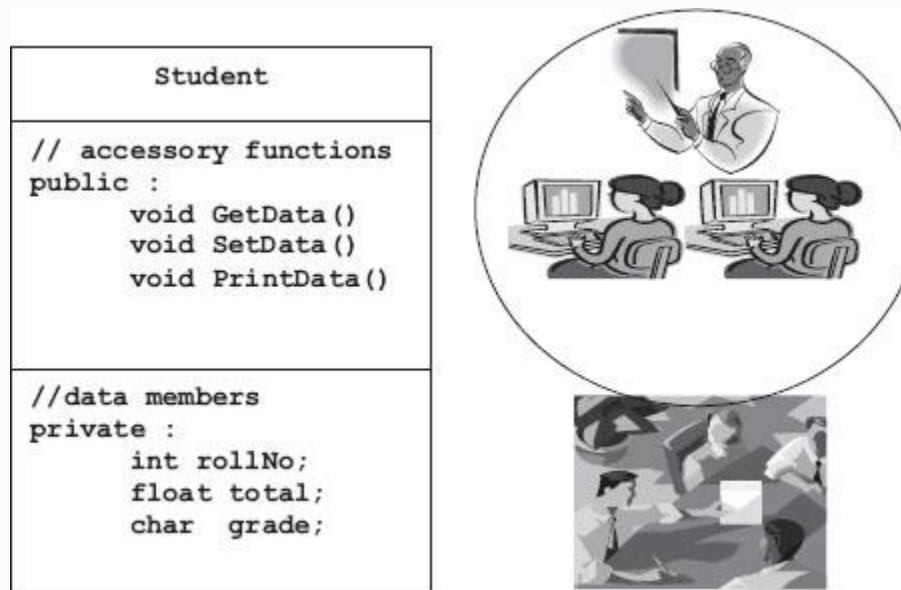
member functions. In this chapter, we will learn how to solve problems by using classes and objects. We will also cover topics such as data hiding and abstraction and access privileges enjoyed by members of the class.

This chapter introduces the concepts of constructor and destructor. Variations in member functions such as friend functions and inline functions are discussed. Classes within a class, called container classes, are introduced. Different methods to use dynamic memory with objects using pointers and references are highlighted with their merits and demerits. *This* operator and static member functions and their relevance and usage are explained.

## 10.2 Classes and Objects

Look around your classroom. There are students all around. First of all, why have they been grouped together by your principal? Firstly, because they all share **common interests**, for example they would like to learn the language C++ or they would like to complete their PG or UG

studies. In computer parlance, we can say that all students have the same functionality. That is why they can be grouped together. Figure 10.1 shows member functions and member data.



**Figure 10.1** Class member functions and attributes

But notice that each student has his own individual attributes. Attributes mean own member data like height, weight, marks, attendance, etc. Also notice that there are about 60 students in your class each with his or her own attributes but common

functionality. We can call 60 instances of **object** of **Students class**. Well so much background for analogy.

**Class:** A collection of objects. We can also define as an array of instances objects. But class can have member functions and member data. Here, unlike array, a class can have different data types as its elements.

**Object:** An object is an entity. That is, it can be felt and seen. Examples are student, pen, table, chair, etc. Therefore, an object is an independent entity that has its own member data and member functions.

**Data Hiding/Data Abstraction.** It is customary to declare all member data as private only. But the data declared as private is hidden and cannot be accessed by any one. This feature is called data hiding or data abstraction. But how can we get them? You can access this only through public member functions. There is no other way. It is comparable to the case where even the chief librarian of a university library cannot take home the books unless he uses the access card supplied by the library.

**Public:** Member functions and data if any declared as **public** can be accessed outside the class member functions.

**Private:** Member data declared as **private** can only be accessed within the class member functions and data is hidden from outside.

**Protected:** Member data and member functions declared as **protected** is private to outsiders and public to descendants of the class in **inheritance** relationship. You will learn more about this in the chapter on Inheritance.

**Encapsulation:** Binding together the member functions and member data with access specifiers like private, public, etc. into object by the class is called **encapsulation**.

**Declaring a class:** Use keyword **class** followed by brace brackets. Inside brace brackets we can include member data and member functions as shown below:

### **Example 10.1: Class Declaration**

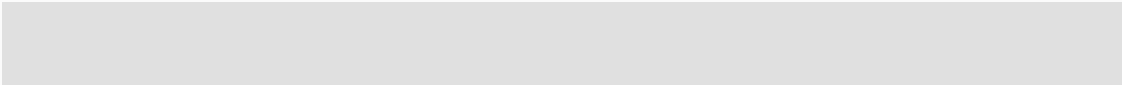
---

```
class Student
{ // constructors and member functions
    public:
        Student()
        {rollNo=50595,name="Anand";}
        ~Student(){}; // Default
    destructor
        int GetRollNo() const { return
rollNo;}
        void SetRollNo ( int n) {
rollNo=n;}
    // member data
    private:
        int rollNo;
};
```

---

### *10.2.1 How to Create an Object to a Class?*

Remember that an object is a variable of a class. Therefore, you can define the object just like you define a variable of a data type. In the example that follows, we create an ordinary object called std and 50 instances of class Student, as well as a pointer that creates an instance of Student on the Heap memory.



## Example 10.2: Creation of an Object of a Class

```
Student std; // std is an object of
class Student
Student std[50]; // There are 50
instances created for Student class
Student *ptr = new Student; // object on
free store created
```

### *10.2.2 How to Access Member Data and Member Functions?*

Once the object to a class is created, we can access the member functions and public member data of a class using `dot ( . )` operator. We have learnt that member data is declared as private in C++ to achieve data hiding and abstraction. Hence, to access these private member data, we need to define public member functions. We call them public accessory functions. To draw an example from our life, you would have observed that books in your library can only be borrowed if you have the library

cards. So is the case with the chief librarian; though he is the custodian of the books, he also needs library cards to take books home.

### Example 10.3: Accessing Member Functions and Member Data

```
Student Std: // object is created
Std.SetRollNo(8345); // Sets roll no to
8345
cout<<Std.GetRollNo(); // displays roll
number
```

In our next example, we will show all the concepts we have discussed so far through a class declaration called Polar. In this program, you will learn the concept of class and object, public accessory functions, etc. The class we will consider is vector in Cartesian form denoted by real number **a** and an imaginary **component b**. In Polar coordinates, the same can be represented by vector whose magnitude is **r** given by the



formula  $r = \sqrt{a^2 + b^2}$ , and the direction is given by  $\theta = \tan^{-1}(b/a)$ .

Our program will accept the real value **a** and imaginary component **b** through a public accessory function called `GetData()` and convert it to Polar form with magnitude **r** and angle  **$\theta$**  through a public accessory function called `ConvertToPolar()` and display the result through a function called `DisplayPolar()`. A word or two about the way we will declare and define functions. If the function to be implemented is big code, then we would declare the prototype inside the class as

---

```
void ConvertToPolar(); // to convert to
polar form and define the function code
outside the class as shown below:
void Polar::ConvertToPolar()
{ double x=0.0;
  r=sqrt(a*a + b*b);
  x=atan(b/a); // x is in radians
  t= (7.0/22.0)*180.0*x; // conversion
to degrees .PI radians 180 degrees
}
```

---

Note that we have used the operator `::` called scope resolution operator to inform the compiler to link up with function prototype declared within the class. Suppose the function to be implemented is one line or two lines only; we can then declare it as inline function as we have explained in [Chapter 3](#).

---

```
inline double GetMag() const {  
    return r ;}
```

---

There is also an easier option of including the code in class definition itself as we have shown: `double GetMag() const {  
 return r ;}`. Then compiler treats the function as inline automatically.

### **Example 10.4: Polarform.cpp Accessing Member Functions and Data of a Class**

---

```
#include<iostream>
#include<cmath> // for maths related
functions like sqrt,cos, tan, tan- etc.
using namespace std;
// Declaration of class called Polar
class Polar
{ // public accessory functions
    public:
        void GetData();
        double GetReal() const { return a;}
// returns real component a. const
implies

// function can only return a, but it
cannot alter
        double GetImag() const { return b;}
        double GetMag() const { return r;} //
Two function GetMag() & GetTheeta
        double GetTheeta() const { return t;}
// for Polar form
        void ConvertToPolar(); // to convert
to polar form
        void DisplayPolar(); // to display in
polar and cartesian forms
// all member data is declared as
private
private:
        double r; // magnitude
        double t; // angle theeta
        double a; // real
        double b; // imaginary
```

```

};

void Polar::GetData() // :: is called
scope resolution operator. It is used
since we are

// defining
GetData() outside the class definition
{ cout<<"Enter Cartesian form Vector
Details "<<endl;
  cout<<" Enter real <a> : ";
  cin >>a;
  cout<<" Enter Imaginary <b> : ";
  cin >>b;
  r=0.0; t=0.0;
}

void Polar::DisplayPolar()
{ cout<<"\n Inside the classes member
function ..."<<endl;
  cout<<"\n Vector in polar form using
r and t directly: magnitude = "<<r
  <<" Angle ="<<t<<endl; // We can
directly use r and t because it is
accessed

// inside the member function of the
class thus equal to public
  cout<<"\n Vector in polar form using
v1.GetMag() etc: magnitude = "<<GetMag()
  <<" Angle ="<<GetTheeta()<<endl;
}

void Polar::ConvertToPolar()
{ double x=0.0;
  r=sqrt(a*a + b*b);

```

```

        x=atan(b/a); // x is in radians
        t= (7.0/22.0)*180.0*x; // conversion
to degrees .PI radians equals 180
degrees
    }
void main()
{   Polar v1; // v1 is the object of
Polar class
    // obtain data for v1
    v1.GetData(); // This is the way we
will call member functions using object
v1
    // convert to Polar form
    v1.ConvertToPolar();
    cout<<"\n Vectors in Polar form ... \n"
<<endl;
    v1.DisplayPolar();
    //cout<<"\nVector in polar form :
magnitude & Angel = "<<v1.r<<v1.t<<endl;
    //Error `r' `t': cannot access
private member declared in class
`Polar'. Commented out
    cout<<"\n\n\n Outside the class using
v1.Getmag() etc..."<<endl;
    cout<<"\n Vector in polar form :
magnitude = "<<v1.GetMag()
    <<" Angle ="<<v1.GetTheeta()<<endl;
// This is the correct way
} // end of
/*Output : Enter Cartesian form Vector
Details
Enter real <a> : 3

```

```

Enter Imaginary <b> : 4
Vectors in Polar form form ...
Inside the class ...
Vector in polar form using r and t
directly: magnitude = 5 Angle =53.1087
Vector in polar form using v1.GetMag()
etc: magnitude = 5 Angle =53.1087
Outside the class using v1.Getmag()
etc...
Vector in polar form: magnitude = 5
Angle =53.1087
*/

```

---

## **A few points to remember**

Inside member functions that is invoked by object you can access the private members directly. For example, in function `DisplayPolar()`, we have used `cout<<r`. This is ok.

But outside, like in main, we cannot use `v1.r`. We have to use public accessory function like `v1.GetMag()`.

### *10.2.3 Constructors and Destructors*

When you need to construct a house, you go to a specialist called architect or mason so that they make the house ready for

occupation and usage. So is the case with Class. When a class is declared and an object is created, the constructor for the class is called. His job is to create the object, allocate memory for the data members and initialize them with initial values if supplied by the user.

The constructor will have the same name as that of the class but no return value. The constructor needs to be declared as public.

```
Student () { } // Default constructor.
```

No initial values. If you do not declare the default constructor, compiler automatically creates it. Hence, the term default constructor.

Overloading means the same function name but different types or different numbers of arguments. For example:

---

```
Student ( int n, char * p) { rollNo=n,  
name = p ; } is an overloaded  
constructor.
```

---

**Destructor:** When the program has ended there is a need to clear the memory

resources allocated to it. This job is done by the destructor.

`~Student(){ };` // Default destructor. The default destructor is called automatically when the program ends.

**Copy Constructor:** This is the second of the special constructors, the first being default constructor, created automatically, if you do not create on your own. It takes one argument i.e., the object to be copied. Look at the following declaration and definition:

---

```
Student( const Student & src)
:rollNo(src.rollNo),name(src.name){}
```

---

The object is passed as constant reference so that it cannot be altered. Once declared, we can use it to copy the entire state of the object into new object.

---

```
Student std; // object is created
Student std1(std) ; // entire object std
is copied on to std1
```

---

Let us attempt a problem to make our understanding of constructors and



destructors clearer. In this program, you will learn the concept of class constructor overloaded constructor, copy constructor, default destructor, etc.

## **Example 10.5: classbasic2.cpp**

### **Constructors and Destructors of a Class**

```
#include<iostream>
using namespace std;
//declare a class called Student
class Student
{   public:
    // overloaded constructors
    Student() {rollNo=0,name="No Name";}
    Student(int n, char *p)
{rollNo=n,name=p;}
    Student( const Student & src) //
copy constructor
    {
rollNo=src.GetRollNo() ;name=src.GetName (
);
        cout<<" Copy
Constructor..."<<endl;}
```

```

        ~Student(){}; // Default destructor
        // public access functions
        int GetRollNo() const { return
rollNo;}
        void SetRollNo ( int n) {
rollNo=n;}
        char * GetName() const { return
name;}
        void SetName( char *p) { name=p;}
private:
int rollNo;
        char *name; // pointer to name
};
void main()
{ Student std; // object created
        cout<<" \n Students rollNo set by
Default constructor : "
        <<std.GetRollNo()<<endl;
        cout<<"\n Students name set by
Default constructor : "
        <<std.GetName()<<endl;
        //set roll no to 6060 and name to
gautam.
        //We have to use public accessory
function to update private data
        // Why ? Because we are not inside
member function
        std.SetRollNo(6060);
        std.SetName("Gautam");
        cout<<"\n Students rollNo set by
SetRollNo : "
        <<std.GetRollNo()<<endl;

```

```

        cout<<"\n Students name set by
SetName : "
        <<std.GetName()<<endl;
    // Now let us make use of overloaded
constructor
    Student std2(7070,"Ramesh");
    cout<<" \n Students rollNo set by
(int n, char *p) overloaded
    constructor : "
        <<std2.GetRollNo()<<endl;
    cout<<"\n Students name set by (int
n, char *p) overloaded : "
        <<std2.GetName()<<endl;
    // Now let us use copy constructor
and copy std2 onto std3
    Student std3(std2); // std is copied
on to new object called std3
    cout<<" \n Students(std3 rollNo) :
"<<std3.GetRollNo()<<endl;
    cout<<"\n Students (std3 name) :
"<<std3.GetName()<<endl;
}
/*Output : Students rollNo set by
Default constructor : 0
Students name set by Default constructor
: No Name
Students rollNo set by SetRollNo :6060
Students name set by SetName :Gautam
Student(std2.rollNo) set by (int n, char
*p) overloaded constructor : 7070
Student(std2.name) name set by (int n,
char *p) overloaded : Ramesh

```

Copy Constructors

```
Students(std3 rollNo) : 7070
```

```
Students (std3 name) : Ramesh*/
```

---

We have explained the salient features in the running program itself. Also take a final look at the public accessory functions we have used.

---

```
int GetRollNo() const { return rollNo;}
```

```
char * GetName() const { return name;}
```

---

When we do not want the function to alter value, we declare as constant as shown above. It will be an error if we write `int GetRollNo() const { return rollNo++;}`. Whereas in the functions shown below `SetName` can set, i.e. alter the name. Hence we have not declared as constant.

---

```
void SetName( char *p) { name=p;}
```

```
void SetRollNo ( int n) { rollNo=n;}
```

---

Look at the constructor declaration and definition we have declared and defined as

---

```
Student(int n, char *p)
{rollNo=n,name=p;} .
```

---

We can also declare and define it as

---

```
Student(int n, char *p) : rollNo(n),
name(p) { } . We will be using this mode
of declaration and definition in all our
programs henceforth.
```

---

## 10.3 Friend Function

Due to data encapsulation and data hiding features of C++, only public member functions can access private data. We can also declare member functions to access private data, like we have been using `GetData()` in our programs, but it is cumbersome. Is there a method by which we can access private data through a non-member function? Friend operator just achieves that. You can include a non-member inside a class by declaring it as friend.

For example, let us revisit the Polar class, declare friend functions for Multiplication and DisplayData() and see ourselves how the program gets simplified.

---

```
friend void DisplayPolar(Polar v); // to
display in polar forms
friend Polar Multiply(Polar v1, Polar
v2);
```

---

When defining these friend functions outside the class, there is no need to use scope resolution operator:: since friend functions are non-members.

---

```
void DisplayPolar(Polar v) // For a
friend we do not use scope resolution
operator
{// We can directly use r and t because
DisplayPolar() is a friend function
    cout<<"Magnitude"<<v.r<<" Angle
="<<v.t<<endl;
}
Polar Multiply(Polar v1, Polar v2)
{
    Polar v3;
    v3.r = v1.r * v2.r; // multiply
the magnitudes
    v3.t = v1.t+v2.t; // add the
angles
```

```
        return v3;
    }
```

---

Also notice that though the functions are **non-members**, because they are **friend functions** we could use private data directly without using public accessory functions. What is then the cost paid for this simplification? You have given access to private data to a non-member function.

### **Example 10.6: classbasic3.cpp**

#### **Friend Functions of a Class**

---

```
/* In this program, you will learn the
   concept friend operator.
   Two functions Multiply and DisplayData
   are declared as friend functions*/
#include<iostream>
#include<cmath> // for maths related
functions like sqrt,cos, tan, tan- etc
using namespace std;
// Declaration of class called Polar
class Polar
```

```

    { //public accessory functions
        public:
            Polar():r(0.0),t(0.0){} // Default
constructor
            Polar( double f, double
g):r(f),t(g){} // This is how we will
declare

// constructor
    ~Polar(){} // Default Destructor
    friend void DisplayPolar(Polar v);
// to display in polar forms
    friend Polar Multiply(Polar v1,
Polar v2);
    // all member data is declared as
private
    private:
        double r; // magnitude
        double t; // angle theeta
};

void DisplayPolar(Polar v) // For a
friend we do not use scope resolution
operator
{ // We can directly use r and t because
DisplayPolar() is a friend function
    cout<<"Magnitude"<<v.r<<" Angle
="<<v.t<<endl;
}

Polar Multiply(Polar v1, Polar v2)
{
    Polar v3;
    v3.r = v1.r * v2.r; // multiply
the magnitudes

```



```

        v3.t = v1.t+v2.t; // add the
angles
        return v3;
    }
void main()
{Polar v1(5.0,53.0),v2(6.0,28.0),v3; //
v1,v2,v3 is the object of Polar class
// multiply Polar Vectors v1 & v2 and
store it in v3
    v3=Multiply(v1,v2);
    cout<<"\nThe Given Vectors in Polar
form form ... \n"<<endl;
    cout<<"\nPolar form vector v1 :";
    DisplayPolar(v1);
    cout<<"\nPolar form vector 2 v2 :";
    DisplayPolar(v2);
    cout<<"\nv3=v1*v2 in Polar Form :";
    DisplayPolar(v3);
} // end of main
/*Output :The Given Vectors in Polar
form form ...
Polar form vector v1 :Magnitude5 Angle
=53
Polar form vector 2 v2 :Magnitude6 Angle
=28
v3=v1*v2 in Polar Form :Magnitude30
Angle =81*/

```

---

## 10.4 Class Within a Class: Container Class

Container class is one of the techniques provided by C++ to achieve reusability of the code. Inheritance is another powerful feature for reusability which we will explore in the chapter on inheritance.

Container class means a class containing another class or more than one class. For example, a computer **has** a microprocessor in it. Further, a Student class can have a data member belonging to a string class. Then we would say that the string class is contained in the Student class. In other words, it can also be called **composition or aggregation** of string class. The advantage of containment is that we can use string class within Student class to store the details of name and address, etc., belonging to string class. Similarly, if we define a class called Date with day, month and year information, we can define object of Date within a class called Student and define Date-related data such as DOB, DOJ, etc.

**Composition is a "has"-type of relation.** For the example we have given above Student has name of string class and DOB of Date class.

## **Example 10.7: PointCircle.cpp An Example for Container Class, i.e. Class as a Data Member of Another Class. Constructors and Destructors of a Class**

In this program, you will learn the concept of class within a class. That is, class as a data member of another class. We will define a class called Point and later declare this as a data member of class called Circle. We will then find the area of the circle.

```
#include<iostream>
#include<cmath>
using namespace std;
// Declare a class called Point
class Point
{
    public:
    Point(){} // default constructor
    ~Point(){} // default Destructor
    float GetX() const { return x;}
    void SetX(float f) { x=f;}
    float GetY() const { return y;}
```

```

        void SetY(float f) { y=f;}
private:
        float x;
        float y;
};
class Circle
{ // We will use user defined data
  called Point declared above
  public:
        Circle(float r, float p, float q
);
        ~Circle(){} // default Destructor
        Point GetCircleCenter() const {
return CircleCenter;}
        void SetCircleCenter(Point f) {
CircleCenter=f;}
        float GetCircleRadius() const {
return CircleRadius;}
        void SetCircleRadius(float f) {
CircleRadius=f;}
        float GetArea() const; // Fn to
Find Area of the Circle
  private:
        float CircleRadius;
        Point CircleCenter; //
CircleCenter is a object of class Point
};
Circle::Circle(float r, float p, float
q)
{
        CircleCenter.SetX(p) ; //
CircleCenter is object of Point

```

```

class
CircleCenter.SetY(q) ;
CircleRadius=r;
}
float Circle::GetArea() const{return
(22.0/7.0)*CircleRadius*CircleRadius;}
void main()
{ Circle circ(3.0,1.5,2.5); // object is
created with center(1.5,2.5)
and Radius=3.0
float area=circ.GetArea();
cout<<" area of the circle with radius
= "<<circ.GetCircleRadius()
<<"
Center("<<circ.GetCircleCenter().GetX()
<<","<<circ.GetCircleCenter().GetY()
<<")="<<area<<" Sq Units"<<endl;
}/*Output:
area of the circle with radius = 3
Center(1.5,2.5)=28.2857 Sq Units*/

```

---

Observe the cout statement we have used **circ.GetCircleCenter()** returns a Point data type. With this Point we have invoked **GetX()** and **GetY()**.

---

```

cout<<" area of the circle with radius
= "<<circ.GetCircleRadius()
<<"
Center("<<circ.GetCircleCenter().GetX()

```

```
<<" , "<<circ.GetCircleCenter() .GetY()  
<<")="<<area<<" Sq Units"<<endl;
```

---

As another example consider a class called Student containing another class called Date.

Date class has date information as data members and public accessory function to set and get date fields such as day, month and year. Class Student declares variables belonging to class Date as shown below:

---

```
Date dateDOB;  
    Date dateDOJ;
```

---

Member functions to access these private variables are also declared. It may be noted that GetDOB() returns variable of type Date.

---

```
Date GetDOB() const { return dateDOB;}  
Date GetDOJ() const { return dateDOJ;}  
void SetDOB( Date dtb) { dateDOB=dtb;}  
void SetDOJ( Date dtj) { dateDOJ=dtj;}
```

---

## **Example 10.8: DateStudent.cpp**

### **Student Class Contains Date Class.**

### **Constructors and Destructors of a Class**

```
#include <iostream>
using namespace std;
class Date
{public:
    Date ( int d=0, int m=0,int y=0)
:dd(d),mm(m),yy(y) { } // constructor
    void SetDate( int d, int m, int y){
dd=d,mm=m,yy=y;}
    int GetDD() const { return dd;};
    int GetMM() const { return mm;};
    int GetYY() const { return yy;};
private:
    int dd;
    int mm;
    int yy;
};
class Student
{ public:
    Student();
    ~Student(){}; // Default destructor
    int GetRollNo() const { return
```

```

rollNo;}

    void SetRollNo ( int n) { rollNo=n;}

    Date GetDOB() const { return
dateDOB;}

    Date GetDOJ() const { return
dateDOJ;}

    void SetDOB( Date dtb) {
dateDOB=dtb;}

    void SetDOJ( Date dtj) {
dateDOJ=dtj;}

private:
    int rollNo;
    Date dateDOB;
    Date dateDOJ;
};

Student::Student()
{ dateDOB.SetDate(14,11,54);
  dateDOJ.SetDate(25,11,77);
  rollNo=5050;
}

void main()
{ Student std; // object created
  cout<<" \n Students rollNo set by
Default constructor : "<<std.
GetRollNo()<<endl;
  cout<<"\n date of Birth :";
  cout<<std.GetDOB().GetDD()
<<"/"<<std.GetDOB().GetMM()
<<"/"<<std.GetDOB().GetYY()<<endl;
  cout<<"\n date of Joining :";
  cout<<std.GetDOJ().GetDD()
<<"/"<<std.GetDOJ().GetMM()

```



```
<<"/"<<std.GetDOJ().GetY()<<endl;  
}  
/*Output : Students rollNo set by  
Default constructor : 5050  
date of Birth :14/11/54  
date of Joining : 25/11/77*/
```

---

Note that in cout statements in the main program, `std.GetDOJ()` returns data type of Date. Hence, with that variable we have further invoked `GetMM()`.

## 10.5 Objects and Data Members on Heap Memory

We have learnt that Heap memory is dynamic memory and we can derive lot of advantage by using heap memory. The main advantage is the conservation of primary memory that arises as a result when we load objects, use and free the memory after use. In this way we can solve large complex problems. So naturally we have to equip ourselves with the skills for deploying objects onto heap memory. As an example, we can declare a class called Student. We

can further declare a pointer to that class and create an object of Student on Heap Memory.

---

```
Student *ptr - new Student;// object  
on free store created
```

---

We use in direction operator -> to access the object functions and member data on the heap. : cout<<ptr->GetName() <<endl;  
// access member data from heap. After use we need to free the heap memory allocated.  
:delete ptr; // we need to delete and free dynamic memory. The following program makes things clear.

**Example 10.9: objheap.cpp**  
**Creation of Object on to Heap**  
**Memory and Access Member Data.**  
**Constructors and Destructors of a**  
**Class**

---

```
#include<iostream>
using namespace std;
//declare a class called Student
class Student
{public:
    Student()
{rollNo=50595,name="Anand";}
    ~Student(){}; // Default destructor
    int GetRollNo() const { return
rollNo;}
    void SetRollNo ( int n) {
rollNo=n;}
    char * GetName() const { return
name;}
    void SetName( char *p) { name=p;}
private:
    int rollNo;
    char *name;
};

void main()
{ Student std; // object created
    Student *ptr = new Student;//
object on free store created
    cout<<" \n Students rollNo set by
Default constructor : "
    <<ptr->GetRollNo()<<endl;
    cout<<"\n Students name set by
Default constructor : "
    <<ptr->GetName()<<endl; // access
member data from heap
```

```

        //set roll no to 50
        ptr->SetRollNo(6060);
        ptr->SetName("Gautam");
        cout<<"\n Students rollNo set by
SetRollNo : "
        <<ptr->GetRollNo()<<endl;
        cout<<"\n Students name set by
Default constructor : "
        <<ptr->GetName()<<endl;
        //now delete the memory from heap
        delete ptr; // we need to delete
and free dynamic memory
    }
/*output :Students rollNo set by Default
constructor : 50595
Students name set by Default constructor
: Anand
Students rollNo set by SetRollNo :6060
Students name set by Default constructor
:Gautam*/

```

---

In the above example, we have created entire object on heap memory. We can also create the member data on heap memory and use indirection operator `->` to access data members. Note that heap memory can be accessed only through the use of pointers. Therefore, our declaration of member data would be pointers

---

```
int *rollNo; // we have to use pointers
as we want to use Heap Memory
char *name;
```

---

Our public accessory functions have to deal in pointers :

---

```
int GetRollNo() const { return *rollNo;}
```

---

Creation of object onto heap memory and access of members would be

---

```
Student *ptr = new Student;// object on
heap created
cout<<ptr->GetRollNo()<<endl;
```

---

**Example 10.10: objmemheap.cpp**  
**Creation of Object and Member**  
**Data on to Heap Memory and**  
**Access Member Data Using**  
**Indirection Operator -> Member**  
**Data. Constructors and Destructors**  
**of a Class**

```

#include<iostream>
using namespace std;
//declare a class called Student
class Student
{ public:
    Student();
    ~Student(){delete rollNo;delete
name;} // Default destructor
    int GetRollNo() const { return
*rollNo;}
    void SetRollNo ( int n) {
*rollNo=n;}
    char * GetName() const { return
name;}
    void SetName( char *p) { name=p;}
private:
    int *rollNo; // we have to use
pointers as we want to use Heap Memory
    char *name;
};

    Student::Student(){rollNo= new
int(50);name= new char;name="ramesh";}
void main()
{ Student *ptr = new Student;// object
on heap created
    cout<<" \n Students rollNo set by
Default constructor : "
        <<ptr->GetRollNo()<<endl;

```

```

        cout<<"\n Students name set by
Default constructor : "
        <<ptr->GetName ()<<endl;
        //set roll no to 6060
        ptr->SetRollNo (6060) ;
        ptr->SetName ("Gautam") ;
        cout<<"\n Students rollNo set by
SetRollNo :"
        <<ptr->GetRollNo ()<<endl;
        cout<<"\n Students name set by
Default constructor :"
        <<ptr->GetName ()<<endl;
    }//end of main
    /*Output : Students rollNo set by
Default constructor : 50
Students name set by Default constructor
: ramesh
Students rollNo set by SetRollNo :6060
Students name set by Default constructor
:Gautam*/

```

---

## 10.6 This Pointer

We are aware that memory management of C++ allocates separate memory area called code section for your code. Accordingly your function code will be in this area. But we also know that variables are stored either in stack area or in heap memory. When a

function is invoked by an object, there is a need to link up function that is called with the objects data. Therefore, the object is forwarded to function as an argument to that function. But this argument, i.e. address of the object invoking the function, is not visible like ordinary formal arguments and it is hidden. This hidden address is called ***this* pointer**. It is called ***this*** because it points this object, i.e. object that has invoked the function.

In a function if you use *this pointer* explicitly then it refers to object member data. Note that we can get the same effect by using object instead. But note that **this** is a pointer and hence it has all the advantages of instant and fast access to object. Look at the program at Example 10.9 rewritten using **this** pointer.

**Example 10.11: this.cpp Object on to Heap Memory Using this Pointer. Constructors and Destructors of a Class**



```
#include<iostream>
using namespace std;
class Student
{ public:
    Student()
{rollNo=50595,name="Anand";}
    ~Student(){}; // Default destructor
    int GetRollNo() const { return
this->rollNo;}
    void SetRollNo ( int n) { this-
>rollNo=n;}
    char * GetName() const { return
this->name;}
    void SetName( char *p) { this-
>name=p;}
private:
    int rollNo;
    char *name;
};

void main()
{
    Student std; // object created
    Student *ptr = new Student;//
object on free store created
    cout<<" \n Students rollNo set by
Default constructor : "
        <<ptr->GetRollNo()<<endl;
    cout<<"\n Students name set by
Default constructor : "
        <<ptr->GetName()<<endl; //
```

```

access member data from heap
    //set roll no to 50
    ptr->SetRollNo(6060);
    ptr->SetName("Gautam");
    cout<<"\n Students rollNo set by
SetRollNo : "
        <<ptr->GetRollNo()<<endl;
    cout<<"\n Students name set by
Default constructor : "
        <<ptr->GetName()<<endl;
    //now delete the memory from heap
    delete ptr;
}
/*Output :Students rollNo set by Default
constructor : 50595
Students name set by Default constructor
: Anand
Students rollNo set by SetRollNo :6060
Students name set by Default constructor
:Gautam*/

```

---

## 10.7 Pointers vs Objects: Use of Constant Declarations

We are aware that pointers are addresses and provide us a fast access to memory and data manipulation. But there are issues of safe operation and data integrity. In Section 4.5.11, we have seen the use of pointers and

effect of declaration of const pointer and constant data. In this section, we will study the effect and efficacy of using const declaration to pointers while working with objects. While manipulating objects stored in memory, a programmer faces several demanding situations such as data should not be altered. In such cases, we would have declared that data as constant.

The pointers to be constant i.e. they should not be reassigned. In such cases, we would declare as const pointers.

Thus, there are two combinations involved, **i.e. *Pointer type and Data Type***. Each type can have two variations of data type: constant and normal. Therefore, we can have a total of four variations in declaration of variables. These are shown below:

---

#### **Pointer to Object**

```
// normal pointer & Normal object on  
heap  
Student *ptr = new Student; // ptr is  
normal pointer on heap
```

#### **Pointer to const Object**

```
// conststudent is a pointer to constant  
Student
```

```
const Student *conststudent = new
Student;
Const pointer to Object
//constptr is constant pointer to
Student
Student * const constptr = new Student;
Const pointer to a const Object
//cptrcstd is a constant pointer to
constant student
Const Student * const cptrstd = new
Student;
```

---

In the example that follows we would demonstrate the methods and rules for deployment of different types of pointers and data type discussed above. When we use normal pointer to normal objects on heap, we can use `Get()` and `Set()` functions. Usage of `Set()` function implies that we can modify the objects data. Note that all `Get()` functions we would have declared `Get()` as `const`. so that it cannot alter the data values but it can only browse these values

---

```
Student *ptr = new Student;
int GetRollNo() const { return rollNo;}
// only read . Function cannot alter
```

```
data.  
void SetGrade( char p) { grade=p;}
```

---

When we use pointer to constant object, we cannot use `serGrade()` functions as it sets data for grade. This is so because we have declared the object as constant.

---

```
// conststudent is a pointer to constant  
Student  
const Student *conststudent = new  
Student;  
ptr->SetGrade('C'); // NOT OK
```

---

When we use the third option, i.e. `const` pointer to constant object, pointer cannot be reassigned nor can the data be amended.

---

```
// const pointer to constant object  
Const Student * const cptrstd = new  
Student;  
int GetRollNo() const { return rollNo;}  
// Allowed  
ptr->SetGrade('C'); // NOT OK
```

---

## **Example 10.12: ptrcdata.cpp To Demonstrate Use of Constant Pointer and Constant Object. Constructors and Destructors of a Class**

```
#include<iostream>
using namespace std;
//declare a class called Student
class Student
{ public:
    Student() {rollNo=50595,grade='A';}
    ~Student(){}; // Default destructor
    int GetRollNo() const { return
rollNo;}
    void SetRollNo ( int n) {
rollNo=n;}
    char GetGrade() const { return
grade;}
    void SetGrade( char p) { grade=p;}
private:
    int rollNo;
    char grade;
};
void main()
{ // ptr is normal pointer on heap
```

```

        Student *ptr = new Student;
        // conststudent is a pointer to
constant Student
        const Student *conststudent = new
Student
        //constptr is constant pointer to
Student
        Student * const constptr = new
Student;
        const Student * const cptr = new
Student;
        cout<<"explore the possibilities
with normal heap pointer *ptr"<<endl;
        cout<<" \n Students rollNo set by
Default constructor : "
                <<ptr->GetRollNo()<<endl;
        cout<<"\n Students Grade set by
Default constructor : "
                <<ptr->GetGrade()<<endl;
        //set roll no to 50 and Garde to B
        ptr->SetRollNo(6060);
        ptr->SetGrade('B');
        cout<<"\n Students rollNo set by
SetRollNo() function : "
                <<ptr->GetRollNo()<<endl;
        cout<<"\n Students name set by
setgrade() function:"
                <<ptr->GetGrade()<<endl;
        cout<<"\nextplore the possibilities
with pointer *conststudentto"<<endl;
        cout<<" constant object
Student"<<endl;

```

```

        cout << "\n Display Grade
..."<<conststudent ->GetGrade()<<endl;
        //conststudent->SetGrade('C');
//NOT OK.

        /*const pointer calling non-const
fn . This is error. conststudent is a
pointer to constant Student, whereas
SetGrade is a non-constant function and
can change value. Hence we cannot use.
You will observe error C2662: 'SetGrade'
: cannot convert 'this' pointer from
'const class Student' to 'class Student
&' Hence it
shown in comments */
        // constant pointer to constant
data object
        cout<<"\nextplore the possibilities
with const pointer *cptr to"<<endl;
        cout<<" constant object
Student"<<endl;
        cout<<"\ncptr->
GetGrade().."<<cptr->GetGrade()
<<endl;//ok
        //cout<<"\ncptr-
>SetGrade('C').."<<cptr->SetGrade('C')
<<endl;//not ok
        //now delete the memory from heap
delete ptr;
delete cptr;
delete conststudent;
}

        /*Output :explore the possibilities

```



```
with normal heap pointer *ptr
  Students rollNo set by Default
constructor : 50595
  Students Grade set by Default
constructor : A
  Students rollNo set by SetRollNo()
function :6060
  Students name set by setgrade()
function:B
  explore the possibilities with pointer
*conststudentto
  constant object Student
  Display Grade ...A
  explore the possibilities with const
pointer *cptr to
  constant object Student
  cptr-> GetGrade() ..A/*
```

---

## **So why do we use constant pointers?**

We do so because constant pointers cannot call non-constant member functions, i.e. member functions that can alter member data. So our member data is safe. While const pointer to constant object is possible, it is tedious and cumbersome. Instead, the same can be achieved more elegantly if we use reference. Remember, insinuations

wherein we cannot reassign pointers, reference is a better alternative.

## 10.8 Passing of Objects by Reference and Pointers to a Function

We are aware that normal variables can be passed to and from function with pass by value and pass by reference. We have also learnt in Chapter 6 on pointers that passing by reference involves passing either by pointer or reference (address). Objects, because they occupy good amount of memory, have to be passed as call by reference. More so because if you use call by value, a copy is made when an object is passed on to function and another copy of the object is made when function returns an object to calling function. What a waste of precious memory! Not only that, other resources are also employed such as calling a copy constructor and destructor. Your program is busy doing these unnecessary jobs when you use call by value.

In our next example, we will study the implications of call by value, call by

reference on objects calling functions. For this, we would declare three different functions outside the class and show the costs and efficiencies involved in different modes of invoking the function.

---

```
Student FindGrade( Student std); //call
by value
Student * FindGrPtr(Student *std); //
call by ref using pointers
Student & FindGrRef (Student &std); //
call by reference using reference
```

---

Also note that we have declared all the member data as pointers.

---

```
int *rollNo; float *total; char *grade;
```

---

Constructor allocates memory in Heap memory with new operator as shown below

---

```
Student::Student(int n, float t, char c)
{ cout<<"\n This is
Student(int,float,char) constructor...\n";
rollNo= new int(n);
grade= new char(c);
total=new float(t);
}
```

---

Destructor is called whenever we have to destroy the object. But remember, our data members like `rollNo`, `grade`, and `total` are also residing on the heap memory. Therefore, we have to explicitly delete them

---

```
~Student() {cout<<"\n This is  
destructor..\n";  
delete rollNo;delete grade;delete total;  
} // destructor
```

---

Copy constructor gets data items from `src` and allocates space on heap with `new` operator.

---

```
Student::Student(Student &src) // src  
means source  
{ cout<<"This is copy constructor..\n";  
rollNo=new int(*(src.rollNo));  
grade=new char(*(src.grade));  
total=new float(*(src.total));  
}
```

---

`src` is the name we use to denote that it is a source file. Generally left-hand side is reserved for current object, i.e. target and source is placed on the right-hand side. We

are creating data members on heap memory. Therefore, `rollNo` and `grade` and `total` have been allocated space using `new` operator. Note that as we are within member function, i.e. constructor, we could use `*src.rollNo` directly without going through public accessory function ( `*src.GetRollNo()` )

**We have created an object of Student**

---

```
Student std; // an object made  
.Therefore, constructor will be called
```

---

**As our data members are pointers, cout statements in the main program are**

---

```
cout<<"\n RollNO:"<<*(std.GetRollNo())  
<<"\t"  
        <<"Total:"<<*(std.GetTotal())  
<<endl;
```

---

When we use call by value by statement `FindGrade(std);` copy constructors are invoked twice: once for forwarding the object to function and once for return from the function. Normally changes made in the

function are not reflected in the calling function in case of call by value. But in this case we are returning the object to called function by statement : `return std;` hence grade changes are reflected in the main program.

A call to a function `FindGrPtr (Student *std)` means we are resorting to pass by reference using pointer. In this case, copy constructors and destructors are not involved at all . This means that primary memory and computer time is conserved. As this is pass by reference using pointer, value set by local variable (`grade`) is reflected in the main. As we are using pointer to object and data members are all pointers we have to use indirection operator to access data members : `cout<<* (std2->GetGrade())`

---

```
Student *std2=new Student(); // std2 is  
created on heap .  
std2=FindGrPtr(std2); // pass by  
reference pointer method  
cout<<"\n Students Grade set by  
FindGrPtr() : "
```

```
<<* (std2->GetGrade() )<<endl;  
//pass by ref ptr
```

---

A call to a function `Student & FindGrRef (Student &std);` means that we are resorting to pass by reference using `&` operator. In this case, copy constructors and destructors are not involved at all. This means that primary memory and computer time is conserved. As this is pass by reference using address, value set by local variable(grade) is reflected in the main.

---

```
Student *std2=new Student(); // std2 is  
created on heap .
```

```
std2=FindGrPtr(std2); // pass by  
reference pointer method  
cout<<"\n Students Grade set by  
FindGrPtr() : "
```

```
<<* (std2->GetGrade() )<<endl;  
//pass by ref ptr
```

---

**Example 10.13: refptr.cpp To Demonstrate Passing of Objects by**

# Reference and Pass by Pointers and Pass by Value. Constructors and Destructors of a Class

```
#include<iostream>
using namespace std;
//declare a class called Student
class Student
{   public:
    Student();
    Student(int n, float t, char c);
    Student(Student &);
    ~Student(){cout<<"\n This is
destructor..\n";
        delete rollNo;delete grade;delete
total;} // Default destructor
    int *GetRollNo() const { return
rollNo;}
    void SetRollNo ( int n) {
*rollNo=n;}
    char *GetGrade() const { return
grade;}
    void SetGrade( char p) { *grade=p;}
    float * GetTotal() const { return
total;}
    void SetTotal( float t) { *total=t;}
private:
    int *rollNo;
```



```

        float *total;
        char *grade;
};

    Student::Student()
    { cout<<"\nThis is default
constructor...\n";
        rollNo= new int;
        grade= new char('D'); //
default Grade
        total=new float;
    }
    Student::Student(int n, float t,
char c)
    { cout<<"\n This is
Student(int,float,char) constructor...\n";
        rollNo= new int(n);
        grade= new char(c);
        total=new float(t);
    }
    Student::Student(Student &src) //
src means source
    { cout<<"This is copy
constructor..\n";
        rollNo=new int(*
(src.rollNo));
        grade=new char(*(src.grade));
        total=new float(*
(src.total));
    }
// Three non-member functions
Student FindGrade( Student std); //call
by value

```

```

Student * FindGrPtr(Student *std); //
call by ref using pointers
Student & FindGrRef (Student &std); //
call by reference using reference
void main()
{ cout<<"\n create an ordinary object
called std for Student class..\n";
    Student std; // an object made.
Therefore constructor will be called
Student *std2=new Student(); // std2 is
created on heap.
std.SetRollNo(50);
    std.SetTotal(99.0);
cout<<"\n RollNO:"<<*(std.GetRollNo())
<<"\t"
    <<"Total:"<<*(std.GetTotal())<<endl;
// now call a function FindGrade(Student
std)
// See how copy constructors are
involved one for forward copy and one
for return
// see also how value set by local
variable(grade =A) is reflected in the
main
// because you are returning the object
by value using statement return std
// you should also see two destructors
    FindGrade(std);
    cout<<"\n Students Grade set by
FindGrade() : "
        <<*(std.GetGrade())<<endl;
// now call a functon FindGrPtr( Student

```

```

*std)
// See how copy constructors and
destructors are not involved at all
// see also how value set by local
variable(grade) is reflected in the main
    std2=FindGrPtr(std2); // pass by
reference pointer method
    cout<<"\n Students Grade set by
FindGrPtr() : "
        <<*(std2->GetGrade())<<endl;
//pass by ref ptr
// now call a function FindGrref(Student
& std)
// See how copy constructors and
destructors are not involved at all
// see also how value set by local
variable(grade) is reflected in the main
    FindGrRef(std); // pass by
reference reference method
    cout<<"\n Students Grade set by
FindGrRef() : "
        <<*(std.GetGrade())<<endl; //
pass by reference - reference */
    }// end of main
// Define functions
Student FindGrade( Student std)
{ cout<<"\n FindGrade( Student std)
function ...\n";
    cout<<" \n Students Grade set by
FindGrade() : \n";
    std.SetGrade('A');
    cout<<"\n inside FindGrade...:"<<*

```

```

(std.GetGrade())<<endl;
    return std;
}
Student * FindGrPtr(Student *std)
{
    std->SetGrade('B');
    cout<<"\n Inside FindGrPtr() :
"<<*(std->GetGrade())<<endl;
    return std;
}
Student & FindGrRef ( Student &std)
{ std.SetGrade('C');
    cout<<"\n Inside FindGrRef() : "<<*(
(std.GetGrade())<<endl;
    return std;
}
/*Output :This is copy constructor..
FindGrade( Student std) function ...
Students Grade set by FindGrade() :
inside FindGrade...:A
This is copy constructor..
This is destructor..
This is destructor..
Students Grade set by FindGrade() :D
Inside FindGrPtr() : B
Students Grade set by FindGrPtr() : B
Inside FindGrRef() : C
Students Grade set by FindGrRef() :C
This is destructor..*/

```

---

## 10.9 Constant Pointers and Constant References

Recall that in the above example function `Student * FindGrPtr(Student *std);` receives the object by reference using pointers. It also amends the Grade. What if we want the function to receive the object by reference because of efficiency and yet should not be able to change the values. Clearly the solution is to pass a constant pointer to Student and thus `FindGrPtr ()` not amend the object. This is because of the stipulation that a constant pointer can only call a constant function and constant function by definition cannot alter the values.

---

```
const Student * const FindGrPtr(const
Student * const std);
const Student & FindGrRef (const Student
&std);
```

---

Call by ref using constant pointers and also constant object. The second call involves call by reference using `&` operator and also

constant object Student. Look at the program presented below to understand the concepts involved better.

### **Example 10.14: constrefptr.cpp To Demonstrate Constant Pointer and Constant Reference and Constant Object. Constructors and Destructors of a Class**

```
//constptrref.cpp
#include<iostream>
using namespace std;
//declare a class called Student
class Student
{ public:
    Student();
    Student(int n, float t, char c);
    Student(Student &);
    ~Student(){cout<<"\n This is
destructor..\n";
    delete rollNo;delete grade;delete
total;} // Default destructor
    int *GetRollNo() const { return
rollNo;}
```

```

        void SetRollNo ( int n) { *rollNo=n;}
        char *GetGrade() const { return
grade;}
        void SetGrade( char p) { *grade=p;}
        float * GetTotal() const { return
total;}
        void SetTotal( float t) { *total=t;}
private:
        int *rollNo;
        float *total;
        char *grade;
};
Student::Student()
{ cout<<"\nThis is default
constructor...\n";
        rollNo= new int;
        grade= new char('D'); // default
Grade
        total=new float;
}
Student::Student(int n, float t, char c)
{ cout<<"\n This is
Student(int,float,char) constructor...\n";
        rollNo= new int(n);
        grade= new char(c);
        total=new float(t);
}
Student::Student(Student &src) // src
means source
{
        cout<<"This is copy constructor..\n";
        rollNo=new int(*(src.rollNo));

```

```

        grade=new char(*(src.grade));
        total=new float(*(src.total));
    }

    const Student * const FindGrPtr(const
Student * const std);
    const Student & FindGrRef (const
Student &std);
    void main()
    { cout<<"\n create an ordinary objet
called std for Student class..\n";
        Student std; // an object made .
Constructor will be called
        std.SetRollNo(50);
        std.SetTotal(99.0);
        cout<<"\n RollNO : "<<*std.GetRollNo()
<<"\t"<<"Total : "
            <<*std.GetTotal()
            <<"Grade : "<<*std.GetGrade()
<<endl;
        // now call const Student * const
FindGrPtr(const Student * const std);
        FindGrPtr(& std); // pass by
reference pointer method
        cout<<"\n Students Grade after
return from FindGrPtr() : "
            <<*std.GetGrade()<<endl;
        // now call a function : const
Student & FindGrRef (const Student
&std);
        FindGrRef(std); // pass by
reference reference method
        cout<<"\n Students Grade after

```



```

returning from FindGrRef() : "
    <<*std.GetGrade()<<endl;
    } // end of main
    const Student * const FindGrPtr(const
Student * const std)
    { //std->SetGrade('B'); // clearly
error constant pointer to constant obj
    cout<<"\n Inside FindGrPtr() : "
<<*std->GetGrade()<<endl;
    return std;
    }
    const Student & FindGrRef ( const
Student & std)
    { //std.SetGrade('C'); // clearly error
. student is a constant obj
    cout<<"\n Inside FindGrRef() : "
<<*std.GetGrade()<<endl;
    return std;
    }
    /*Output :create an ordinary objet
called std for Student class..
This is default constructor...
RollNO :50 Total :99Grade : D
Inside FindGrPtr() : D
Students Grade after return from
FindGrPtr() : D
Inside FindGrRef() : D
Students Grade after returning from
FindGrRef() :D
This is destructor..*/

```

---

Note that in cout statements in `main()` and functions `std.GetGrade()` return a pointer of character type as per our function prototype declaration. Hence we have used dereference operator `*` : `*std.GetGrade()` or `*std->GetGrade()`;

## 10.10 Static Member Data

Recall our definition of encapsulation. It is binding of member data and member functions into a single entity by class. A class has access rights defined for its data members and enjoys the benefits of protective and security features like data hiding, abstraction, etc. Further you are aware that member data is exclusive to an instance of object. For example, Student Hari cannot share the data of Student Shiva.

If we want a single data item to be available to all instances of the class, one way is to declare it as a global variable. But in that case we would lose all advantages of being a member of a class like data security, access privileges, etc.

For example, if we want a variable called count to keep track of the number of objects being created and the number of objects being deleted, how do we declare such a variable?

If we declare it in global section, we would lose benefits of being within the class. But if we declare it within the class, it becomes part of the class and hence belongs to an instance of class and cannot be shared amongst objects. Then how can the variable keep track of the number of objects being created?

Variable declared as static, though resides inside a class; it can keep track of the objects being created or destroyed. This means that it is accessible to all instances of the class. We can say that static data belongs to class and not to an object.

---

```
class Student
{ public:
    Student() {count++;}
    ~Student() {count--;} // Default
    destructor
    static int count; // static data member
    declared as public
};
```

---

Therefore, to give access to all instances of the class declare the variable as **static**. It does not belong to class and is not initialized along with member data when the object is created. Also note that no memory resources are allocated by static declaration made inside the class. Hence, the static variables are required to be declared globally outside the class and then only resources are allocated.

---

```
int Student::count=0; // mandatory to  
declare in global section
```

---

In the example shown below, we have instantiated four objects, two outside the controlling braces and two within the controlling braces. We can use static variable count with any one of the objects to get the count of objects created:

---

```
Student a,b;  
{ Student ramesh, suresh;  
cout<<"ramesh.count"<<ramesh.count<<"\t"  
<<"suresh.count"  
}
```

---

The life of variables is till the controlling brace brackets. Hence we would get count as 2 when we check the count after the controlling braces.

### **Example 10.15: static1.cpp To Demonstrate Usage of Static Data Within a Class**

---

```
#include<iostream>
using namespace std;
//declare a class called Student
class Student
{ public:
    Student() {count++;}
    ~Student() {count--;} // Default
destructor
    static int count; // static data
member declared as public
};
int Student::count=0; // mandatory to
declare in global section
void main()
{
```

```

{ Student a,b; // two objects have been
created
    Student ramesh, suresh;

cout<<"ramesh.count"<<ramesh.count<<"\t"
<<"suresh.count"
    <<suresh.count<<endl;
    cout<<"The above result shows that
static variable is available to ramesh &
suresh";
}

    cout<<"\nafter brace brackets
:"<<endl;
    cout<<"a.count="<<a.count<<endl;
    Student w,x,y,z;
    cout<<"\nafter creation of w x y z
:"<<endl;

cout<<"a.count"<<a.count<<"\t"<<"b.count
"<<b.count<<endl;
}
/*Output :ramesh.count4 suresh.count4
The above result shows that static
variable is available to ramesh & suresh
after brace brackets :
a.count=2
after creation of w x y z :
a.count6 b.count6*/

```

---

What happens if you declare the static data as private? Quite simple. By now you must have understood that the only way to access a private data is through a public accessory function. Code is reproduced for static data type that is declared as private.

### **Example 10.16: static2.cpp To Demonstrate Usage of Private Static Data Within a Class**

```
#include<iostream>
using namespace std;
//declare a class called Student
class Student
{public:
    Student() {count++;}
    ~Student() {count--;} // Default
destructor
    int GetCount() const { return
count;}// public function to access
static private data
    private:
        static int count; // static data
member declared as private
```

```

};
int Student::count=0; // mandatory
to declare in global section
void main()
{ Student a,b;
  { Student ramesh, suresh;

cout<<"ramesh.count"<<ramesh.GetCount()
<<"\t"<<"suresh.count"
      <<suresh.GetCount()<<endl;
  cout<<"The above result shows
that static variable is available to
ramesh & suresh";
  }
  cout<<"\nafter brace brackets
:"<<endl;
  cout<<"a.count="<<a.GetCount()
<<endl; // because static private
declaration
  Student w,x,y,z;
  cout<<"\nafter creation of w x y z
:"<<endl;
  cout<<"Student.count"<<x.GetCount()
<<"\t"<<"y.count"<<y.GetCount()<<endl;
  }
/*Output :ramesh.count4 suresh.count4
The above result shows that static
variable is available to ramesh & suresh
after brace brackets :
a.count=2
after creation of w x y z :
Student.count :6 y.count : 6*/

```



---

## 10.11 Static Member Functions

The data members declared as static is available for all instances of the class. That is why we could use `x.GetCount()` and `y.GetCount()` in the above program. Therefore, it really does not matter which object calls the function that displays static data.

To make it independent of object we need to define the member function as static.

---

```
static int GetCount() { return count;}//  
public function to access static private  
cout<<Student::GetCount() // you can  
call static function without object  
                                // by using  
class name and scope resolution operator
```

---

The next program illustrates the static function concept and usage.

**Example 10.17: static3.cpp To Demonstrate Usage of Static**

# Member Functions Within a Class

```
#include<iostream>
using namespace std;
//declare a class called Student
class Student
{ public:
    Student() {count++;}
    ~Student() {count--;} // Default
destructor
    static int GetCount() { return
count;} // public function to access
static private data
    private:
        static int count; // static data
member declared as public
};

    int Student::count=0; // mandatory
to declare in global section
    void main()
    { Student a,b;
      { Student ramesh, suresh;

cout<<"Student.count"<<Student::GetCount
()<<endl;
}
cout<<"\nafter brace brackets :"<<endl;
cout<<"Student.count="<<Student::GetCoun
t()<<endl;
```

```

        Student w,x,y,z;
        cout<<"\nafter creation of w x y z
:"<<endl;
        cout<<"Student count
:"<<Student::GetCount()<<endl;
    }
/*Output : Student.count4
after brace brackets :
Student.count=2
after creation of w x y z :
Student count :6
*/

```

---

## 10.12 Summary

1. Class, we use in C++, to define our own data type. Class data types can be of different data types.
2. Object is a variable of data type class.
3. A class therefore allows us to encapsulate member functions and member data into a single entity called object.
4. Encapsulation is binding of data and member function into objects by a class.
5. Access privileges are public, private and protected. Members declared as public are available to all, whereas members declared as private are only available to instances of the class. Protected members are private to all but public to descendants, i.e. derived classes in inheritance relation.
6. The only way to access private member data is through public accessory functions.

7. Use scope resolution operator `::` to define member functions outside the class.
8. Member functions declared and defined inside a class are treated as inline functions, though explicitly not stated.
9. Constructors and destructors have the same name of the class but they have no return types.
10. Friend function carries a keyword `friend`, even if declared inside the class does not belong to class. It has access to all private data members directly without going through the public accessory functions. When defining friend functions outside the class, there is no need to use scope resolution operator as the friend function does not belong to class.
11. Containers are provided by C++ to achieve reusability of code. Containments are also called composition or aggregation. Composition is a 'has' -type of relation.
12. Objects and data members can be allocated space on Heap memory using `new` operator and released using `delete` operator. Indirection operator `->` is used to access members on heap.
13. This pointer is a hidden pointer that is forwarded to a function. It is called this pointer because it refers to this object that has invoked the function.
14. Constant pointers and constant objects are used to ensure data security and integrity. Constant pointers can only call constant functions.
15. Objects are passed by reference, either by reference or through pointers.
16. When using `const` object and no reassignment is involved it is better to use reference method that is easy to use and elegant rather than `const` pointer to constant object.
17. Static member data though resided inside a class does not belong to class . It is available to all instances of the

class. It is mandatory to define in global section static member data so that memory resources can be allocated.

18. To make invoking of member functions independent of objects we need to declare public functions handling static data as static functions.

## Exercise Questions

### Objective Questions

1. A constructor is called whenever

1. An object is declared
2. An object is used
3. A class is declared
4. A class is used

2. A class having no name

1. Is not allowed
2. Cannot have a constructor
3. Cannot have a destructor
4. Cannot be passed as argument

3. Constructors can take arguments

1. Zero arguments
2. Two arguments
3. No arguments
4. Any number of arguments

4. Destructors can take arguments T

1. One argument
2. Two arguments
3. No arguments
4. Any number of arguments

5. Constructor can be overloaded while destructor cannot be overloaded True/False

6. Constructors are used to create data members. True/False
7. Constructors return void data type. True/False
8. Member functions declared and defined inside a class are inline though not explicitly defined. True/False
9. In C++, a function declared within a class is called
  1. Inline function
  2. Member function
  3. Contained function
  4. Class function
10. The default access specifies for data members in C++ is
  1. Public
  2. Private
  3. Protected
  4. Inherited
11. Declaring a friend function inside a class allows the friend function to access
  1. Private data of all member objects of the class
  2. Private member functions of all member objects of the class
  3. Both a and b
  4. None of the above
12. Static variable declared as private is available to all instances of the class
  1. Through public function
  2. Through public function declared as static
  3. Directly without public function
  4. None of above
13. 13) A static function can handle
  1. Non-static data
  2. Only static data
  3. Private data
  4. Public data.
14. Class declaration and definition and is placed above void main(). Which one of the following statements are NOT

true?

1. It is required by all functions
2. It is a global declaration
3. Space available
4. They are global declarations and available to all

15. Const pointer can call

1. Non-constant function
2. Const function
3. Any function
4. None

16. If object is declared constant then this pointer is a constant pointer True/False

17. Dynamic memory can be allocated with

1. Pointers only
2. Reference operator only
3. Both a and b
4. None

18. Constant this pointer can call

1. Non-constant function
2. Const function
3. Any function
4. None

#### **Short-answer Questions**

19. Distinguish between private and public access specifiers.
20. Distinguish a class and an object. Why is scope resolution operator employed in C++?
21. Explain data hiding and encapsulation.
22. Explain why private variables cannot be accessed outside the class directly.
23. What is the solution provided by language C++? For problem at 4.
24. Explain constructor overloading.
25. Explain the difference between default constructor and constructor with arguments.

26. What is a friend function? How is it useful?
27. Explain static data variable.
28. Distinguish a static data and a global data.
29. What is a container class? Explain with examples.
30. Explain the use of this pointer.
31. Why are constant pointers deployed?
32. Explain the context under which constant pointer to constant object can be deployed.
33. Constant pointer can only invoke a constant function. Why this restriction?
34. How do you create objects on heap memory? Explain with examples.
35. How do you create member data on heap memory? Explain with examples
36. Differentiate Const pointer to data and pointer to constant data.

#### **Long-answer Questions**

37. Declare a class called Account with data members, `accNo`, `name`, `balance`. Declare a data members as private and member functions as public. Declare a member function called Transact which will allow withdrawal of amount and then display the balance.
38. Rewrite the above code with a private member function called `DisplayTrDate()` that will be invoked by withdraw function and will display transaction date.
39. Implement a class called `FractionNo` with suitable constructors and destructors. Include functionality like add, multiply and inverse.
40. Declare a Point class inside a class Rectangle and calculate the area of the rectangle.
41. Write code for creating objects of two employees with `idNo`, `name`, `basicPay`. Demonstrate the use of constructor, copy constructor.



42. Rewrite the code for sl 5 with object and member data on the heap memory.
43. Write a class based cpp to calculate the grades of n number of students. There are 2 mid-term examinations. Each mid-term examination consists of one online for 20 marks and one subjective paper for 20 marks. For internal marks consider the best of the online and the best of subjective papers. External papers to have 60% weight. Declare grade A for 80% and above, Grade B for 60 to 79%, Grade C for others. Use a private member function to determine the grade of the student.
44. Declare a class called a rational number, defined as two integer variables called numerator and denominator, and include functionality for +, -, \*, / of two rational numbers. Use friend function for mathematical operations.
45. Implement a class to represent a Circle that holds radius, x, y(center coordinates) and functions to calculate area and circumference. Use containment concept.
46. Use a class called counter. Declare a private static variable called count. Use it through public function and display the no. of objects created. Constructor and destructor must keep track of count addition on creation and subtraction on deletion.

### **Solutions to Objective Questions**

1. a
2. a
3. d
4. c
5. True
6. False

- 7. False
- 8. True
- 9. b
- 10. b
- 11. c
- 12. b
- 13. b
- 14. c
- 15. b
- 16. True
- 17. a
- 18. b

# 11

## C++ Special Features

### ERRORS AND EXCEPTIONS AND OPERATOR OVERLOADING

#### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to*

- Write objective-oriented C++ programs that incorporate errors and exceptions.
- Write programs with your own errors and exceptions class to suit particular and individual needs.
- Write programs with overloading of operators.

#### 11.1 Introduction

Many of you would have come across a pop-up while using the Internet or operating system, wherein the pop message says “OS or Browser has experienced a serious problem and needs to be closed down. Error reporting is in process”. Indeed, error has occurred. Despite elaborate testing prior to delivery, errors cannot be prevented but they can be minimized. In this chapter, we will study of mechanism of C++ to handle errors and exceptions.

An important feature of any modern OOPs language is overloading. Overloading means making a function or operator perform more than one task, depending on the arguments supplied, at run-time. Function overloading is one such example. C++ has overloaded the operators too. Intrinsic operators have been overloaded thus enhancing the efficiency of language.

## 11.2 Errors and Exceptions

There are three types of **errors** that can crop up in a program:

- **Syntactical Errors:** Can be easily detected and corrected by compilers.
- **Logical Errors:** Arise due to the programmer not understanding flow of logic.
- This can be corrected by extensive testing.
- **Bugs:** These can be fixed by the programmer using `assert()` macros and debuggers available.

**Exceptions**, on the other hand, are unusual conditions that occur at run-time and can lead to errors that can crash a program. The exceptions can be classified as **Synchronous**, i.e. those can be predicted. For example, array index going outside the permissible values can be easily detected because such errors occur only when you have executed the statement involving array index. The synchronous exception can occur at any one or more of the following situations:

- Memory allocation exception
- IO handling exception
- Maths handling exception like division by zero

**Asynchronous exceptions**, those that cannot be predicted.

Generally, the resources required for the program are allocated at the very beginning and resources are demanded at run-time.

Thus, it is likely that our program may exceed the initial allotment. As an example, consider allocated memory for an array. When such an exception occurs, we need a mechanism to carry such information to the area where resources are allocated so that we can take corrective actions and prevent program crashing.

### 11.3 Try and Catch Blocks

C++ raises an exception object. For it to do so, we need to use a try block, wherever we expect likely exception. Catch blocks that follow try blocks catch the exception object and take remedial action.

Consider the case of allocating dynamic memory allocation for a two-dimensional matrix. Try will allocate dynamic memory. If it fails due to non-availability of memory, then it will throw the exception object. Catch block will catch the exception thrown by try block and take remedial measures. In the example that follows, we will show how to use try and catch blocks for allocation of memory for a two-dimensional matrix.

## Example 11.1: Use of Try and Catch Block. Allocation of Memory for a Two-dimensional Matrix

```
1. try
2. { A=new int * [m];
3. for (int i=0;i<m;i++)
4. A[i]=new int[n];
5. }catch (bad_alloc) { cout<<"\n bad
allocation"<<endl;exit(1);}
```

## Example 11.2: A Try Block can Catch Multiple Exceptions Like IO Exception, Index Out of Bounds Exception, etc

```
try
{
```

```
// allocation code here
}  
catch (bad allocation object){ }  
catch ( File IO exception object) { }
```

---

### **Example 11.3: Nesting of Try Block**

```
try  
{ // allocation code here  
    try{// inner try block code here}  
    catch ( File IO exception object)  
{ }  
}  
catch (bad allocation object){ }  
catch ( File IO exception object) { }
```

---

## **11.4 Exception Classes**

In the following example, we will demonstrate the use of exception handling in the case of divide by zero error that occurs at run-time. The example is that of



calculation of density using the formula  $\text{density} = \text{mass}/\text{area}$ . The inputs are mass and radius. At run-time, if the radius is entered as zero, the program throws an exception object called `DivideByZero()` ;

Note that we have defined a class `DivideByZero{ }` with an empty body so that we can use it as a tool to catch the exception and enter the catch block. In the catch block, we will inform the user and make a decent exit through `exit(0)` statement.

### **Example 11.4: dividebyzero.cpp A Program to Demonstrate Exception Class – Divide by Zero**

---

```
1.  #include<iostream> // for using cin
    and cout for input and output library
2.  using namespace std;
3.  class DivideByZero{} // exception
    class used to enter catch
4.  template <class T>
```

```

5.  T FindDensity(T mass , T radius);
6.  void main()
7.  { double mass, radius, area,
density;
8.  while (1)
9.  { cout << "\n Enter mass in Kgs :";
10. cin>> mass;
11. cout << "\n Enter radius < 0 to test
divide by zero and exit> :";
12. cin>> radius;
13. try
14. {if ( radius==0.0)
15. throw DivideByZero();
16. else
17. {area = 4*3.141581
*radius*radius;density = mass/area;}
18. }catch ( DivideByZero)
19. { cout<<"\n Dive by zero error has
occurred :"<<endl;
20. cout<<"\n the parameters are : mass
="<<mass
21. <<"\t"<<"radius ="<<radius<<endl;
22. cout<<"\n Exiting the
programme:"<<endl;
23. exit(0);
24. }
25. cout<<"\n the parameters are : mass
="<<mass
26. <<"\t"<<"radius
="<<radius<<"\t"<<"density
="<<density<<endl;
27. cout<<"\n Normal execution of the

```

```

programme."<<endl;
28. }
29. } // end of main
/*Output: Enter mass in Kgs :25.00
Enter radius < 0 to test divide by zero
and exit> :3.0
the parameters are : mass =25 radius =3
density =0.221049
Normal execution of the programme.
Enter radius < 0 to test divide by zero
and exit> :0.0
Dive by zero error has occurred :
the parameters are : mass =45 radius =0
Exiting the programme:*/

```

**L** **class DivideByZero{};** When an exception  
**i** object is thrown, control passes to catch block,  
**n** wherein corrective mechanisms are in place. To  
**e** gain entry into catch block, we have declared  
and defined an exception class inside the main  
**N** class. Note that this exception class has no body.  
**o** Its sole purpose is to give entry into catch block.

.  
**3**  
:

**L** **throw DivideByZero();** is inside a try block.  
**i** This means that try block throws

**n** DivideByZero() object.

**e**

**N**

**o**

**.**

**1**

**5**

**:**

**L** are try block and Lines No 18–24 are catch  
**i** block.

**n**

**e**

**s**

**N**

**o**

**.**

**1**

**3**

**–**

**1**

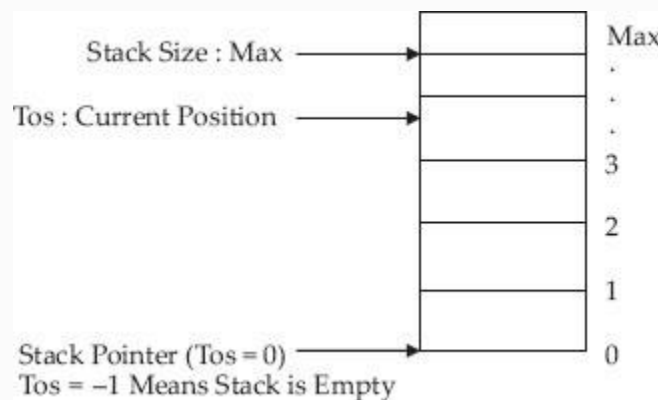
**8**

**:**

We will use stack data structure to demonstrate errors and exception features. But before we do that, there is a need to understand the data structure called stack.

## 11.5 Stack Data Structure

Stack is a linear data structure. Data is inserted and extracted based on a technique called Last in first Out(LIFO) concept. As an example think of an activity of students submitting assignments to their teacher. The teacher will receive and pile them in a stack on the table and will start corrections from the top. Therefore, last in assignment gets service first. As you will see, computer systems depend on stack structure to solve several problems. Representation is shown in Figure 11.1.



**Figure 11.1** Stack representation and terms

# Stack Operations

- Create a stack.
- Check if stack is full or empty. We cannot insert into a full stack. Neither can we extract from an empty stack.
- Initialize the stack. For example,  $\text{tos} = -1$ , and  $\text{tos} = \text{max} - 1$ ; are initialization commands. Stack extends from 0 to  $(\text{max} - 1)$ .
- Push (insert) an element onto stack if it is not full.
- Pop(extract) an element from the stack if it is not empty.
- Read from top of stack.
- Display/print the entire stack.

As an example, while programming stack operations, we can include two exceptions, namely, `stackfullexception` and `stackemptyexception`.

---

```
class Stack
{ public: .....
    .....
    // Exception classes
        class stackfullerror {};
        class stackemptyerror {};
};
```

---

Try block and its associated catch block can be placed in the main program, where push and pop operations are contemplated. As shown below, try

---

```
{
    s.push("Thunder");
    cout<<s.top()<<endl;
    s.push("Anand");
    cout<<s.top()<<endl;
    s.push("Ramesh");
}
```

---

The following programs, 11.6, will highlight these features:

### **Example 11.5: stackassign2.cpp Implements the Stack ADT Using Dynamic Arrays**

---

```
1. #include<iostream>
2. #include<string>
3. using namespace std;
4. const int max_size=3;
5. class Stack
6. { public:
7.     Stack(); //default constructor
8.     ~Stack(); //destructor
9.     Stack& operator=(const Stack&);
```

```

//Assignment operator overloading
10. int size() const; //returns number
of elements in stack
11. int empty() const; //returns 1 if
stack is empty
12. int & top(); //returns top of stack
13. void pop(); //pops top of stack
14. void push(const int &); //pushes
element on top of stack
15. // exception classes
16. class stackfullerror {};
17. class stackemptyerror {};
18. private:
19. int* stack_array; //dynamic array
20. int tos; //top of stack pointer
21. int stack_size; //size of dynamic
array
22. };
23. //default constructor
24. Stack :: Stack()
25. { stack_array = new int[max_size];
26. stack_size = max_size;
27. tos = -1;
28. }
29. //destructor
30. Stack:: ~Stack() {delete []
stack_array;}
31. Stack& Stack :: operator=(const
Stack& S) // Assignment Operator
32. { stack_array = new
int[S.stack_size];
33. tos = S.tos;

```



```

34. stack_size = S.stack_size;
35. for(int i =0 ; i<S.size(); i++)
36. stack_array[i] = S.stack_array[i];
37. return * this; // for chaining
purpose
38. }
39. int Stack :: size() const {return
tos+1;}
40. int Stack :: empty() const
41. { if(tos == -1)
42. return 1;
43. else return 0;
44. }
45. void Stack:: push(const int & val)
46. {if( tos<stack_size-1 )
47. { tos++;
48. stack_array[tos] = val;
49. }
50. else
51. throw stackfullerror();
52. }
53. void Stack :: pop()
54. { if(tos >= 0)
55. tos--;
56. else
57. throw stackemptyerror();
58. }
59. int& Stack :: top()
{return(stack_array[tos]);}
60. void main()
61. { Stack s;
62. //check for stack full exception

```

```

63.  try
64.  {s.push(10);s.push(20); s.push(30);
65.  }catch(Stack::stackfullerror)
66.  { cout<<"\nstack full exception.
Cannot be pursued on to stack:"<<endl;
67.  cout<<"\nExiting the catch block of
stackfull exception .."<<endl;
68.  }
69.  //overloaded assignment operator
70.  cout<<"\n Result of = operator on
stack objects s & t .."<<endl;
71.  Stack t;
72.  t=s; // assignment Operator
73.  cout<<"\n Elements of Stack s
"<<endl;
74.  while(!s.empty())
75.  { cout<<s.top()<<endl;s.pop();}
76.  cout<<"\n Elements of Stack t after
= operator"<<endl;
77.  while(!t.empty())
78.  { cout<<t.top()<<endl; t.pop();}
79.  }//end of main
/* Output: Result of = operator on stack
objects s & t ..
Elements of Stack s : 30 20 10
Elements of Stack t after = operator 30
20 10*/

```

---

--	--

<b>Line No. 5:</b>	Declare a class for Stack class.
<b>Lines No. 10 and 11:</b>	declare two functions called <code>size()</code> and <code>empty()</code>
<b>Lines No. 12 and 13:</b>	declare two functions <code>top()</code> and <code>pop()</code>
<b>Lines No. 16 &amp; 17:</b>	Define <code>class stackfullerror {}</code> and <code>class stackemptyerror {}</code> inside class Stack
<b>Line No. 19:</b>	declares a dynamic array called <code>stack_array :int * stack_array;</code>
<b>Line No. 40:</b>	<code>int Stack :: empty() const //</code> <code>empty() function definition</code>
<b>Lines No. 63–65:</b>	Try block wherein we have tried to push elements on to stack
<b>Lines No. 65–68:</b>	catch block catches <code>stackfullerror</code>

## 11.6 Operator Overloading

Predefined operators such as  $+$ ,  $-$ ,  $/$ ,  $*$ , etc. are defined and provided by the compiler to work on intrinsic (basic) data types such as `int`, `char`, `double`, etc. We have also discussed at length while discussing the concept of function overloading that if a function can perform more than one task, then we can call it as overloading and it is an efficient way of utilizing scarce resources like primary memory. Operator overloading also improves the efficiency and throughput of the program by conserving the primary memory of C++.

If we can make these predefined operators work on user-defined data types such as classes, we would call this operator overloading. For example, consider a predefined operator  $+$ , and its normal operation:

---

```
int x=10, y=20;
int z = x + y ; // contents of x and y
are added and placed in z
Now consider two complex numbers in
polar form of representation:
Polar v1(25.0,53.50); // magnitude and
angle theta
```

```
Polar v2(5.0, 45.00);  
Polar v3;
```

---

### *11.6.1 Overloading of + Operator and \* Operator*

Operator + overloading means we can write  $v3 = v1 + v2$ . In that case, what would compiler do? Exactly what we tell the compiler to do in operator overloading function definition like

Convert both polar form vectors into Cartesian coordinates like  $r \cos t + j \sin t$ .

Add real terms of  $v1$  and  $v2$  and place them in  $v3$ .

Add imaginary terms of  $v1$  and  $v2$  and place them in  $v3$ .

Convert the resultant  $v3$  into polar form.

For multiplication:

Convert both into polar form.

Multiply magnitude of `polar1` & `polar2` to get resultant magnitude.

Add angles of `polar1` & `polar2` to get resultant `theta`.

Declare resultant magnitude and resultant `theta` as a result of multiplication.

Let us attempt this interesting problem in our next problem. In this program, we will show how to overload operators `+` and `*` to perform operations on objects `v1` and `v2` so that we can write

---

```
v3=v1+v2;  
v3=v1*v2;
```

---

### **Example 11.6: opplus.cpp A Program to Show Operators `+`, `*` Overloading**

---

```
1. #include<iostream>  
2. #include<cmath>  
3. using namespace std;  
4. class Polar
```

```

5.  { // public accessory functions
6.  public:
7.      Polar() {r=0.0;t=0.0;a=0.0;b=0.0;}
8.      Polar(double x, double y)
        {a=x;b=y;r=0.0;t=0.0;}
9.      double GetA() const{return a;}
10.     double GetB() const {return b;}
11.     double GetR() const{return r;}
12.     double GetT() const {return t;}
13.     void ConvertToPolar(); // to
convert to polar form
14.     void ConvertToCartesian(); // to
convert to Cartesian form
15.     friend void DisplayPolar(const
Polar &v); // to display in cartesian
forms
16.     friend void DisplayCartesian(const
Polar &v);
17.     Polar operator+(const Polar &v1);
18.     Polar operator*(const Polar &v1);
19. private:
20.     double r; //mag
21.     double t; //theeta
22.     double a; // real
23.     double b; // imaginary
24. };
25. void Polar::ConvertToPolar()
26. { double x=0.0;
        r=sqrt(a*a + b*b);
        x=atan(b/a);
        t= (7.0/22.0)*180.0*x; // PI radians
equals 180 degrees

```

```

27. }
28. void Polar::ConvertToCartesian()
29. { double x=0.0;
      x=atan(b/a);
      a=r*cos(x);
      b=r*sin(x);
30. }
31. Polar Polar::operator+ (const Polar
    &v)
32. { Polar v3(a+v.GetA(),b+v.GetB());
    return v3;}
33. Polar Polar::operator* (const
    Polar &v)
34. { Polar v3;
35.     float x=0.0;
36.     v3.r=r*v.GetR();
37.     v3.t=t+v.GetT();
38.     x = t*(7.0/22.0)*180;
39.     v3.a=r*cos(x);
40.     v3.b=r*sin(x);
41.     return v3;
42. }
43. void DisplayCartesian(const Polar &
    v)
44. { cout<< «\n Vector in cartesian :
    real = «<<v.a<<» imaginary
    =»<<v.b<<endl; }
45. void DisplayPolar(const Polar & v)
46. { cout<< "\n Vector in polar form
    etc: magnitude = "<<v.r<<" Angle
    ="<<v.t<<endl;
47. }

```



```

48. void main()
49. { Polar v1(3.0,4.0),v2(3.0,4.0),v3
,v4;
50.     cout<<»\n Given Vector V1 :»»;
51.     DisplayCartesian(v1);
52.     cout<<»\n Given Vector V2 :»»;
53.     DisplayCartesian(v2);
54.     v3=v1+v2;
55.     cout<<"\n after v1 + v2 :operator
overloading .."<<endl;
56.     DisplayCartesian(v3);
57.     v1.ConvertToPolar();
58.     v2.ConvertToPolar();
59.     DisplayPolar(v1);
60.     DisplayPolar(v2);
61.     v4=v1*v2;
62.     cout<<»\n after v1 * v2 :operator
overloading ..»<<endl;
63.     DisplayPolar(v4);
64. }

/* Given Vector V1 :Vector in cartesian
: real = 3 imaginary =4
Given Vector V2 : Vector in cartesian :
real = 3 imaginary =4
after v1 + v2 :operator overloading ..
Vector in cartesian : real = 6 imaginary
=8
Vector in polar form etc: magnitude = 5
Angle =53.1087
Vector in polar form etc: magnitude = 5
Angle =53.1087

```

```
after v1 * v2 :operator overloading ..  
Vector in polar form etc: magnitude = 25  
Angle =106.217*/
```

---

### *11.6.2 Operator << Overloading*

We can write individual functions to display the object, as shown below, through the usage of `DisplayPolar()` function, but this is cumbersome. Instead, can we simply write `cout<<v3` and still expect the same result? Operator << over loading achieves this result.

#### **Example 11.7: opcout.cpp To Demonstrate Overloading of << Operator**

```
1. #include<iostream>  
2. #include<cmath> // for maths related  
   functions like sqrt,cos, tan, tan- etc  
3. using namespace std;  
4. // Declaration of class called Polar  
5. class Polar
```

```

6. { public: Polar():r(0.0),t(0.0){} //
Default constructor
7.     Polar( double f , double
g):r(f),t(g){} // constructor
8.     ~Polar(){} // Default Destructor
9.     Polar operator*(const Polar &v1);
10.    void Display(ostream & src) const
11.    { src<<"Magnitude =
"<<r<<"\t"<<"Angle theeta ="<<t<<endl;}
12.    double GetR() const{return r;}
13.    double GetT() const {return t;}
14.    private:double r; t ; // magnitude
& angle theeta
15. };
16. ostream & operator<<(ostream &src,
const Polar v){ v.Display(src); return
src;}
17. Polar Polar::operator* (const Polar
&v)
18. { Polar v3;
19.     float x=0.0;
20.     v3.r=r*v.GetR();
21.     v3.t=t+v.GetT();
22.     return v3;
23. }
24. void main()
25. { Polar
v1(5.0,53.0),v2(6.0,28.0),v3; //
v1,v2,v3 is the object of Polar class
26.     cout<<"\nThe Given Vectors in
Polar form form ... \n"<<endl;
27.     cout<<"\nPolar form vector v1

```

```

: "<<v1<<endl;
28.      cout<<"\nPolar form vector 2 v2
: "<<v2<<endl;
29.      // multiply Polar Vectors v1 & v2
and store it in v3 using overloaded
operator
30.      v3=v1*v2;
31.      cout<<"\nv3=v1*v2 in Polar Form
: ";
32.      cout<<v3;
33. } // end of main
34. /*Output: The Given Vectors in Polar
form form ...
35. Polar form vector v1 :Magnitude = 5
Angle theeta =53
36. Polar form vector 2 v2 :Magnitude =
6 Angle theeta =28
37. v3=v1*v2 in Polar Form :Magnitude =
30 Angle theeta =81*/

```

---

### *11.6.3 What We Can Overload and What We Cannot Overload*

All operators of C++ can be overloaded. We cannot overload the following operators:

- Member function access operator : .
- Dereferencing pointer to member : \*
- Conditional operator : ?
- Scope resolution operator : ::
- Sizeof() operator

- All operators operating on a class must be overloaded except operators such as assignment = , address operator &, and comma operator , . Precedence of overloaded operators is exactly the same as that of normal operators.

### *11.6.4 Overloading Assignment Operator =*

By now we appreciate what is meant by overloading. If we overload the assignment operator, we should be able to write in our code  $A = B$  where  $A$  and  $B$  are objects. In C++, we are aware that we can write assignment of several variables like

---

```
double x = y = z = 3.141519;
```

---

Clearly after assigning the initial assignment as  $z = 3.141519$ , the operator must return the reference of what is assigned so that chaining can be continued and compiler can assign the values to  $y$  and thence to  $x$ . We also discussed that this pointer is always present when a function is invoked by the object and we can conveniently make use of this pointer to return the reference.

---

```
Polar & Polar :: operator = ( const
Polar & src)
{ r = src.r; t = src.t;
  return *this; } // here we are
returning ref for chaining explained
above
```

---

As a second example, look at the problem in Example 11.6 discussed above wherein we have used operator = overloading

---

```
9 Stack& operator=(const Stack&);
//Assignment operator overloading
31 Stack& Stack :: operator=(const
Stack& S) // Assignment Operator
32 { stack_array = new
int[S.stack_size];
33 tos = S.tos;
34 stack_size = S.stack_size;
35 for(int i =0 ; i<S.size(); i++)
36 stack_array[i] = S.stack_array[i];
37 return * this; // for chaining
purpose
38 }
```

---

<b>Line No.</b>	is a prototype declaration
-----------------	----------------------------

<b>9:</b>	
<b>Line No. 32:</b>	allocates space for new integer array as per S stack size
<b>Line No. 33:</b>	equates S.stack_size to stck_size
<b>Lines No. 35–36:</b>	get the array values
<b>Line No. 37:</b>	returns a reference * this

If we are using templates, the assignment operator for a class called Stack in Example 11.4 looks like the following:

```
template <class T>
Stack<T>& Stack<T> :: operator=(const
Stack& S)
{ stack_array = new T[S.stack_size];
  tos = S.tos;
  stack_size =    S.stack_size;
  for(int i =0 ; i<size; i++)
    stack_array[i] = S.stack_array[i];
  return *this;}
```

## 11.7 Pre- and Post-increment Operator Overloading

While end result is the same, C++ treats pre- and post-increment operators differently.

---

```
For pre-increment operators, the syntax
is: Height & operator ++ (); //pre fix
For post-increment operators, the syntax
is: Height & operator ++ (int); // post
fix
```

---

Note that the argument int shown in case of post-increment operators is a dummy. It is used by compiler just to distinguish from pre-increment operators.

### **Example 11.8: prepostfix.cpp Demonstrates Operator ++ Overloading.ence**

---

```
1. #include<iostream>
2. #include<cmath> // for maths related
   functions like sqrt,cos , tan , tan- etc
```



```

3.  using namespace std;
4.  class Height {
5.  public:
6.  Height(float h);
7.  Height & operator ++ (); //pre fix
8.  Height & operator ++ (int); // post
fix
9.  ~Height(); // Destructor
10. Height(Height const& m); // Copy
constructor
11. Height& operator= (Height const& m);
// Assignment operator
12. float GetHeight() const { return
yourHeight;}
13. private:float yourHeight;
14. };
15. inline Height::Height(float
h):yourHeight (h){}
16. inline Height::~~Height() {} //
Destructor
17. Height & Height::operator++ () {
++yourHeight;return * this;}
18. Height & Height::operator ++ (int)
{yourHeight++;return * this;}
19. void main()
20. { Height
Ramesh(179.0),Gautam(172.0),Anand(171);
// three objects created
21.  cout<<"\n Given Height..."<<endl;
22.  cout <<Ramesh.GetHeight()<<" :
"<<Anand.GetHeight()<<" :
"<<Gautam.GetHeight()<<endl;

```

```

23.    ++Ramesh; ++Anand; ++Gautam;
24.    cout<<"\n Height after prefix ++
overloading..."<<endl;
25.    cout <<Ramesh.GetHeight()<<" :
"<<Anand.GetHeight()<<" :
"<<Gautam.GetHeight()<<endl;
26.    Ramesh++; Anand++; Gautam++;
27.    cout<<"\n Height after postfix ++
overloading..."<<endl;
28.    cout <<Ramesh.GetHeight()<<" :
"<<Anand.GetHeight()<<" :
"<<Gautam.GetHeight()<<endl;
29. }
    /*Output: Given Height...179 : 171 :
172
    Height after prefix ++ overloading...
180 : 172 : 173
    Height after postfix ++ overloading...
181 : 173 : 174*/

```

**Line  
s No.  
17  
and  
18:**

Pre- and post-increment operators definition. Observe that your Height has been increased and \*this (reference object) has been returned

## 11.8 Overloading Operators New and Delete

We are aware from our study of dynamic memory allocation that new operator is used for allocation of memory in heap memory. The syntax for new and delete operators is presented here:

---

```
int *x = new int; // creation and
allocation of heap memory
*x=25; // assign value
Alternately, we can use a single
statement
int *x = new int(12); allocate value
12 to pointer variable on heap memory
.....
delete x; // q pointer has been
deleted
```

```
For array declarations : //x is a
pointer variable pointing to array by
name x
int *x =new int[12]; // having 12
contiguous locations.
.....
delete [] x; // deletion of pointer to
an array
```

---

In the example that follows, we will show overloading of new and delete operators. As

we are overloading new and delete operations, for memory allocation, we cannot use their original definitions. Instead, we use `malloc()` and `free()` functions from C language. The syntax is shown below:

---

```
int *x ; // x is a pointer variable
//allocate dynamic memory space using
malloc()
x = (int *)malloc( 12 *sizeof(int));
.....
free(x); // delete the memory
allocated
```

---

### **Example 11.9: Student1.cpp A program to Overload New and Delete**

---

```
1. #include <iostream>
2. #include <cstdlib> // for using
   malloc() & free operators
3. #include <new> // including new fro
```

```
c standard library
4.  using namespace std;
5.  class Student
6.  {public:
7.    Student() {rollNo = marks = 0;}
    //default constructor
8.    Student(int a, int b) {rollNo =
a;marks = b;}
9.    void Display()
10.   {cout <<"Roll No : "<<rollNo << "
Marks : " <<marks << endl;}
11.   int GetrollNo()const { return
rollNo;}
12.   int Getmarks()const { return marks;}
13.   void * operator new(size_t size);
14.   void operator delete(void *std);
15.   private :
16.     int rollNo, marks;
17. };
18. // overloaded new operator
19. void *Student::operator new(size_t
size)
20. { void *std;
21.   cout << "In overloaded new.\n";
22.   std = malloc(size); // we are
using malloc because we are overloading
new
23.   if(!std)
24.     {bad_alloc e; throw e;}
25.   return std;
26. }
27. // delete operator overloaded
```

```

28. void Student::operator delete(void
    *std)
29. { cout << "In Side overloaded
    delete.\n"; free(std);}
30. void main()
31. { Student *std;
32. try {
33.     std = new Student (50595, 89);
34.     }catch (bad_alloc e) {cout <<
    "Allocation error for obj1.\n";exit(1);}
35. std->Display();
36. delete std; // free an object
37. }
/*Output: In overloaded new.
Roll No :50595 Marks : 89
In Side overloaded delete.*/

```

---

**Li  
n  
es  
N  
o.  
13  
a  
n  
d  
14  
:**

void \* operator new(size\_t size);  
void operator delete(void \*std); are  
prototypes for new and delete operators.

<b>Line No. 19 – 26:</b>	Definition for new operator. Line No 21 uses malloc function of C language for allocation space demanded by size. Also note that at line no 19, size_t is a data type defined in all include definitions supplied by compiler writers. It is equivalent to int.
<b>Line No. 23 – 26:</b>	show use of exception called bad_alloc, which throws an exception object e. The exception object is caught at line No 34.
<b>Line No. 28:</b>	shows overloading of delete operator, which uses free() function of C.

## 11.9 Conversion (Casting) Operators

There are many occasions wherein an object of a class, say class number, needs to be converted to an object of string class.

Conversion operators are useful in this case. It means userdefined objects such as classes and structures, etc., can be type casted to other objects by a facility called conversion operators. Conversion operator has to be a non-static member function (why?). For example:

---

```
Student std; // an object of Student
Operator char * () const; // converts
std into char *
Operator int() const; // converts std
into int
```

---

```
Operator Person() const; //
converts std into an object of user-defined
class Person ;
```

Const implies that conversion operator cannot change the original object. Note also that conversion operator does not have any return type. Compiler automatically converts object to char \* when it sees the



command operator `char *` `()` . Note also that overloaded conversion operators can be used to convert user-defined object into fundamental data type.

### **Example 11.10: `//convop.cpp` A Program to Show Conversion Operator**

```
1.  #include<iostream>
2.  #include<cstring>
3.  using namespace std;
4.  class OwnString
5.  { public:
6.      OwnString();      //default
    constructor
7.      OwnString(const char *s){stg=s;};
    // constructor with argument
8.      operator const char *() {return
    stg;} // conversion operator to convert
    to ctype string
9.  private:
10.     const char *stg;
11.     int size;
12. };

```

```

13. void main()
14. {   OwnString stg("Example of
conversion Operator");
15.     int n = strcmp(stg,"Example");
16.     cout<<"Strings are NOT same :
"<<n<<endl;
17.     n=strcmp(stg,"Example of
conversion Operator");
18.     cout<<"Strings are same :
"<<n<<endl;
19. }
/*Output
Strings are NOT same : 1 : Strings are
same : 0*/

```

<b>Li ne No . 7:</b>	is overloaded constructor
<b>Li ne No . 8:</b>	conversion operator that returns stg of type <code>const char *(ctype)</code> so that we can use it with <code>strcmp</code> in line no. 15 and 17.

## 11.10 Summary

1. Errors are of three types: syntactical errors, logical errors and bugs.
2. Syntactical errors can be easily detected and corrected by compilers.
3. Logical errors arise due to the programmer not understanding the flow of logic. This can be corrected by extensive testing.
4. Exceptions are unusual conditions that occur at run-time and can lead to errors that can crash a program. They can be classified as synchronous and asynchronous exceptions.
5. Synchronous exceptions are those that can be predicted. They occur due to one or more of reasons like memory allocation, IO handling and exceptions such as divide by zero, etc.
6. Asynchronous exceptions are those that cannot be predicted.
7. Try and catch blocks are used to address problems arising out of errors and exceptions. Wherever error or exception is likely, the programmer includes a try block. Try block detects the errors and exceptions and throws error/exception objects. A try block can catch multiple exceptions and it can be nested.
8. Catch block catches error/exception objects and takes remedial actions.
9. Exception objects are used to gain entry into catch blocks.
10. Operator overloading refers to basic operators performing operations on user-defined data types like classes and objects in addition to intrinsic data types.
11. Input and output operators like << and >> can be overloaded to operate on objects in conjunction with cin and cout objects of iostream.

12. The operators that cannot be overloaded are: **dot operator** `(.)`, **\***, **?**, and **::**
13. Conversion operators can be used to convert objects of a class into objects of another class or into fundamental type.

## Exercise Questions

### Objective Questions

1. Which of the following statements are valid function overloading?
1. Same function name with different no of arguments.
  2. Same function name with different types of arguments.
  3. Same function name with same arguments but different return types.
  4. Same function name with different arguments and different return types.
1. i and ii  
2. ii and iii  
3. iii  
4. i and iv
2. Which of the following statements are true about data types of C++ errors and exceptions?
1. Errors can be prevented
  2. Errors can be minimized
  3. Synchronous errors can be predicted
  4. Asynchronous errors can be predicted
1. i and ii  
2. ii and iii  
3. ii, iii and iv  
4. i and iv
3. Which of the following statements are valid with respect to C++ operator overloading?

1. Basic operators work on basic data types.
2. Basic operators work on basic data types and also user-defined data types.
3. All operators of C++ can be overloaded.
4. Overloading of operator is an OOPS feature.

1. i, ii and iv
2. ii and iii
3. iii and iv
4. i and iv

4. Which of the following statements are valid with respect to C++ errors?

1. Synchronous errors can be detected by compiler.
2. Asynchronous errors can be detected by compiler.
3. Logical errors can be detected by compiler.
4. Bugs can be detected by compiler.

1. i, ii and iv
2. ii and iii
3. iii and iv
4. i and iv

5. Which of the following statements are valid with respect to C++ exceptions?

1. Synchronous errors cannot be predicted.
2. Asynchronous errors cannot be predicted by compiler.
3. IO exception is a synchronous exception.
4. Resource allocation is done in the beginning and exception occurs at run-time.

1. i, ii and iv
2. ii, iii and iv
3. iii and iv
4. i and iv

6. Exception objects are thrown by

1. Try block
2. Catch block
3. IO block
4. Exception class

7. Remedial and corrective actions on the exception caught are handled by

1. Try block
2. Catch block
3. IO block
4. Exception class

8. Which of the following operators cannot be overloaded?

1. [ ]
2. ::
3. ->
4. Dereferencing pointer to member \*

1. i, ii and iv
2. ii, iii and iv
3. ii and iv
4. i and iv

9. ? operator can be overloaded                      TRUE/FALSE

10. Bugs are a type of exception                      TRUE/FALSE

11. Which of the following are true in respect of operator overloading?

1. >> & << are stream extraction operators
2. >> & << are left shift & right shift operators
3. both i) and ii)
4. Meaning based on the context

1. i
2. ii
3. iii
4. iii and iv

12. Which of the following statements are true with respect to conversion operators?

1. Conversion operators are stati(c)
2. Objects of one class can be converted to object of another class.
3. Objects of one class cannot be converted to fundamental data types.
4. Fundamental data types can also be converted into object of a class

1. ii and iv
2. ii and iii
3. i and iii
4. iii and iv

### Short-answer Questions

13. Distinguish errors, exceptions.
14. Differentiate synchronous and asynchronous exceptions.
15. What are logical errors?
16. What are bugs? How can they be minimized?
17. Differentiate roles of try and catch blocks.
18. What is the role of exception object?
19. What is operator overloading?
20. Why does OOPS language support operator overloading?
21. Which of the operators cannot be overloaded? And why?

#### **Long-answer Questions**

22. Explain what are errors and exceptions.
23. Explain how error handling is carried out in C++.
24. Explain different modes of usage of try and catch blocks with examples.
25. Explain operator overloading with suitable examples.
26. Explain working of new and delete operator overloading with examples.
27. What is conversion operator? Explain with examples.
28. Compare functions and operator overloading for solving a problem. Which is better and why?

#### **Assignment Questions**

29. Implement a class called array with functionality such as `find()` `sort()` `insert()` and `delete()` with exception class such as `ArrayOutOfBound { }` and `IOException { }`.
30. Write a cpp having an exception class called `Insufficientfund { }`. Your application should raise exception object if there no sufficient funds in the user's bank account. The exception object to give a suitable message.
31. Implement queue data structure with class `QueFull{ }` and class `QueEmpty{ }`. Exception classes. Hint:

Use an array implementation for queue. The elements in the array are serviced on first come first serve basis. Insertion of an element is at the end and removal for service is from the front.

32. Implement a class called Array. Demonstrate the operator overloading for functionality such as addition, multiplication by a scalar, dot product and cross product.
33. Implement a class called Rational with operator overloading for: + , = , != operators.
34. Implement a class called String with all necessary operator overloading.

### **Solutions to Objective Questions**

1. c
2. b
3. a
4. d
5. b
6. a
7. b
8. a
9. False
10. True
11. d
12. a



# 12

## Inheritance

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to*

- Use inheritance to extend a new class from the existing classes.
- Understand the types of inheritances and concepts therein.
- Understand the concepts of ADT, virtual functions, and dynamic data binding.

### 12.1 Introduction

The concept of inheritance is not new to us. We inherit property, goodwill and name

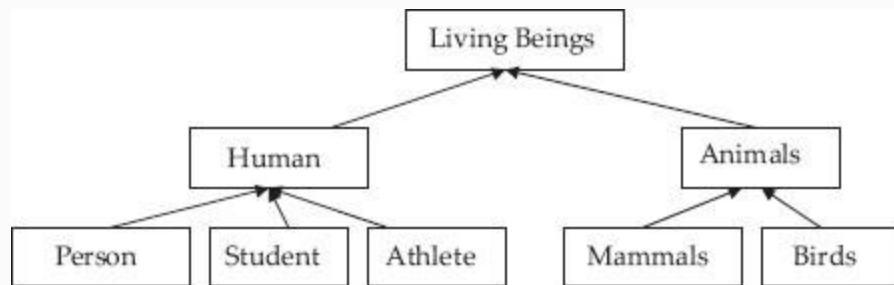
from our parents. Similarly, our descendants will derive these qualities from us.

Reusability of code is one of the strong promises that are made by C++ and inheritance is the tool selected by C++ to fulfill this promise. In this chapter, we take you through inheritance and its properties and rules. We will dwell into the terminology of inheritance and the types of inheritances. We will introduce the reader to overloading and overriding concepts and introduce the various types of inheritances through examples. We will also cover the virtual functions and ADTs in detail. Each concept you will learn through an example.

## 12.2 Inheritance Hierarchy

Study the inheritance hierarchy shown in Figure 12.1. Human and Animal derive qualities from living beings. Professionals, students and Athletes are derived from Human. We say these three categories have inherited from Human. Class Human is called base class and Student and Athlete are called derived classes. What can be

inherited? Both member functions and member data can be inherited. Have you noticed the *direction of the arrow* to indicate the inheritance relation? It is pointed upwards as per modelling language specifications



**Figure 12.1** Inheritance hierarchy

Inheritance specifies *is* type of relation. Observe that a Student is a Human. Similarly Mammal **is** Animal. Human is a **base class** and Student is a **derived class**. Derivation from base class is the technique to implement **is** type of relation. When do we use Inheritance? You inherit so that you can derive all the functionality and member data from base class and derived class can

add its own specialized or individualistic functionality. For example, Athlete derives all functionality and attributes of Human and in addition adds sports and Athletic functionality and attributes on its own.

### **Example 12.1: Syntax for Inheritance**

```
class Human
{ public:
    protected: // variables declared as
protected are public to
    derived classes
// protected variables
};
// Student inherited from Human ,
inheritance type is public
    class Student : public Human
{ public:
private:
};
```

Note that members declared as private are not available to derived class. If you make these member data as public then security is compromised. So what is the way out? Declare the member data as protected. Protected variables mean that these variables are public to derived classes and private to other classes.

If a public member function has the object, i.e. object has invoked the member function, then that function can access *all the public member functions and public member data. Public functions of a class can access all the private member data and member functions of its own class and **protected member data and member functions of their base classes.***

**Example 12.2: inherit1.cpp**  
**Program to Show Inheritance with**  
**Derived Class and Base Class**

---

```
1. #include <iostream>
2. using namespace std;
3. class BaseClass
4. { public:
5. void SetBase(int a, int b) { length =
a; bredth = b; }
6. void DisplayBase(){ cout
<<"length:"<<length << " : "
<<"bredth:"<<bredth <<endl; }
7. protected:
8. int length, bredth;
9. };
10. // public inheritance
11. class DerivedClass : public
BaseClass
12. { public:
13. DerivedClass(int x) { height = x; }
14. void DisplayDerived() { cout
<<"Height:"<< height <<endl; }
15. private:
16. int height;
17. };
18. void main()
19. { // create an briefcaseject of
derived class height=20
20. DerivedClass briefcase(10);
21. briefcase.SetBase(25,30); // access
member of base
22. cout<<"\naccess & display member of
base class"<<endl;
23. briefcase.DisplayBase(); // access
```

```

member of base
24. cout<<"\nAccess & display members of
derived class"<<endl;
25. briefcase.DisplayDerived(); // uses
member of derived class
26. }//end of main
/*Output: access & display member of
base class : length:25 : bredth:30
Access & display members of derived
class : Height:10 */

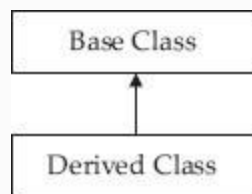
```

<b>Line No. 11</b>	Base class extends i.e. derives from Base class
<b>Line No. 20</b>	A derived Class object briefcase has been created Height=10
<b>Line No. 21</b>	DC, briefcase accesses BC and sets its length and Bredth
<b>Line No. 23 &amp; 25</b>	briefcase displays both base class and derived class data

## 12.3 Types of Inheritance

Inheritance means ability to derive a descendant class from base class. So in how many ways can we do that? We will refer to BC for base class and DC for derived class.

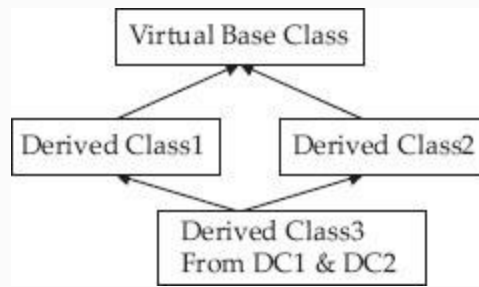
**Single Inheritance:** Figure 12a. DC is derived from the base class



**Figure 12.2a** Single inheritance

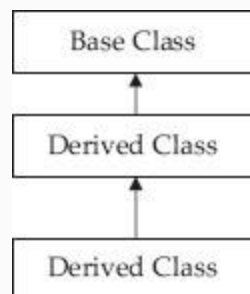
**Multiple Inheritance:** In Figure 12.2b, multiple inheritance is at display. Note that subclass DC# has inherited from DC2 and also from DC1. Base class is virtual because both DC1 & DC2 have only one base class but each derived class feels that it has its own base class. Hence, the term “virtual”. Thus, we can also call this type of inheritance as inheritance with virtual base class.





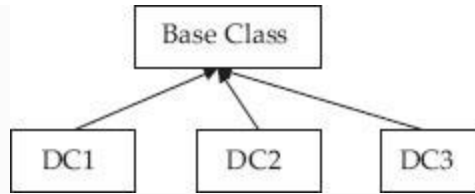
**Figure 12.2b** Multilevel inheritance

**Multilevel Inheritance:** Figure 12.2c is self-explanatory



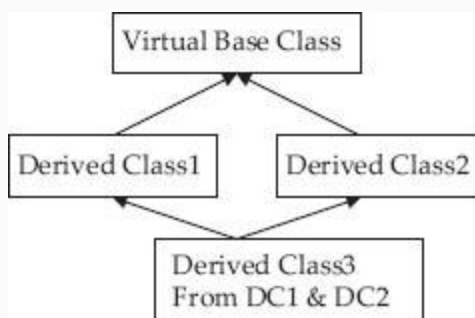
**Figure 12.2c** Multi level inheritance

**Hierarchical Inheritance:** Figure 12.2d is self-explanatory



**Figure 12.2d** Hierarchical inheritance

**Hybrid Inheritance:** Figure 12.2e. DC1 and DC2 are derived from BC. Hence base class is virtual base class. However, DC3 is derived from both DC2 & DC3. DC3 holds properties and data partly from DC2 and DC3 and also from DC as per design. Hence, this type of inheritance is called Hybrid Inheritance.



**Figure 12.2e** Hybrid inheritance

## 12.4 Constructors and Destructors

When we are creating object of derived class like Student, notice that Student constructor will call the base class constructor Human and first constructs Human portion of member data like idNo and name. Then Student constructor will initiate its own part of data like Category called CAT.

Similarly, when an object has to be destroyed, the derived class destructor class will be called first, followed by base class destructor. Notice also that derived classes, in addition to importing the functionality of base class like `Work()` and `Learn()`, add their own functionality like `UseNet()` and `UseGym()`. This is the way Inheritance is to be used. Inherit but add your own functionality in derived classes.

## 12.5 Base Class Function Overriding

Base class Human defines two functions, namely, `Work()` and `Learn()`. Derived class, if it does not define these functions once again, it can use base class `learn()`

or `Work()` . But if it wants its own implementation, different from base class, it can define these functions in its class definition.

---

```
void Learn(){ cout<<"\n Students learn
professional subjects... "<< endl;}
void Work(){ cout<<"\n Students work at
college laboratories... "<< endl;}
```

---

When derived class object calls these functions, it will get derived class functionality only but NOT the base class functionality. If derived class object specifically needs the base class functionality, then it must call base class function using scope resolution operator as shown below:

---

```
cout<<"\n Student does Work() belonging
to base class "<<endl;
std::Human::Work();
```

---

## Example 12.3: inherit2.cpp

### Program with Constructors and Destructors

```
1. #include<iostream>
2. using namespace std;
3. enum CAT { BTECH,MBA,MCA,MSC}; //
Category of student
4. enum BAT { ATHLETE,SPORTS}; // Batch
of Athlete or sports person
5. class Human
6. { public:
7. Human():idNo(0),name("Noname") //Base
constructor
8. {cout<<"\n default Human
constructor.."<<endl;}
9. Human(int n , char
*p):idNo(n),name(p) // Base overloaded
constructor
10. {cout<<"\nHuman
constructor.."<<endl;}
11. ~Human(){cout<<"\nHuman
destructor.."<<endl;} // base destructor
12. // Public functions
13. int GetReg()const{return idNo;}
14. void SetId(int n) { idNo=n;}
```

```

15. char * GetName() const { return
name;}
16. void SetName( char *p) { name=p;}
17. void Learn()const{ cout<<"\n Human
beings strive to learn ..."<<endl;}
18. void Work()const { cout<<"\n Human
work ..."<<endl;}
19. protected:
20. int idNo;
21. char *name;
22. };
23. class Student : public Human
24. { public:
25. Student():Human(),stcat(BTECH)
{cout<<"\nStudent constructor..
"<<endl;}
26. Student(int n , char *p
):Human(n,p),stcat(BTECH)
27. {cout<<"\n Student(int n, char *p)
constructor.."<< endl;}
28. ~Student(){}; // Default destructor
29. CAT GetCAT() const { return stcat;}
30. void SetCAT ( CAT c) { stcat=c;}
31. //students do differently functions
32. //void Learn(){ cout<<\n Students
learn professional subjects..." endl;}
33. //void Work(){ cout<<\n Students
work at college laboratories..." endl;}
34. void Play() { cout<<"\n Students
play at college and at home..."<<endl;}
35. void UseNet(){ cout<<"\n Students
use internet for learning.."<<endl;}

```

```

36. private:
37. CAT stcat; // students category
38. };
39. void main()
40. { Student std(50,"Thunder");
41. std.Learn();
std.Work();std.UseNet();
42. cout<<"\n Students name is
:"<<std.GetName()<<endl;
43. cout<<"\n Explicit call to base
class work....."<<endl;
44. std.Human::Work(); // explicit call
to base class work
45. }
46. /*Output : Human constructor..
47. Student(int n, char *p)
constructor..
48. Human beings strive to learn ...
49. Human work ....
50. Students use internet for
advancement...
51. Students name is :Thunder
52. Explicit call to base class
work.....
53. Human work ....
54. Human destructor..*/

```

<b>Li</b>	<b>Declare enum data type CAT &amp; BAT.</b>
-----------	--

**n  
e  
N  
o.  
2  
&  
3:**

**Li  
n  
e  
N  
o.  
7  
&  
9:**

Base class constructors class Human.

**Li  
n  
e  
N  
o.  
13  
&  
16  
:**

public accessory functions

**Li  
n  
e  
N  
o.**

implements two functions of Human `work()`  
& `Learn()`



17  
&  
18  
:

Li  
n  
e  
N  
o.  
2  
3:

Student class inherits public from Human

Li  
n  
e  
N  
o.  
2  
5:

Student constructor first call base class  
constructor Human to set id No and Name and  
then sets its variables like CAT

Li  
n  
e  
N  
o.  
3  
2  
&  
3  
3:

void Learn() { cout<<\n Students learn  
professional subjects..."endl; }

	void Work(){ cout<<\n Students work at college laboratories...”endl;}
	show that Student over rides base class functions work() and Learn() at Line No. 17 &18.
<b>Line No. 34 &amp; 35:</b>	Student Declares its own functions like Play() & UseNet()
<b>Line No. 44:</b>	Student std(50, "Thunder"); creates an object std of type Student. Firstly it calls base class constructor Human. ( Line No 44), then calls for Student(int , char *) constructor and completes the creation of object.
<b>Line No.</b>	std.Learn(); std.Work();std.UseNet(); are invoked by object std. As we have commented out statement 32 & 33 , learn() and Work() of base class Human are invoked as shown in line 46 & 47.

<b>41</b> <b>:</b>	
<b>Li</b> <b>n</b> <b>e</b> <b>N</b> <b>o.</b> <b>5</b> <b>2</b> <b>&amp;</b> <b>5</b> <b>3:</b>	Show explicit call to base class over ridden function

If you remove comment from 33 & 34 and recompile and run you will see Student objects `Learn()` and `Work()` will be invoked since they have overridden base class functions. Human constructor.

```
Student(int n, char *p) constructor..
Students learn professional subjects...
Students work at college laboratories
Students use internet for advancement...
Students name is: Thunder
Human destructor.
```

In our next example, we would work out two derived classes, namely, Student and Athlete

### **Example 12.4: inherit3.cpp**

#### **Program to Show Inheritance with Two Derived Classes**

Copy Line No. 1 to 38, i.e. up to declaration of Students class prior to statement 1 and compile.

---

```
1. class Athlete : public Human
2. { public:
3.   Athlete():Human(),atcat(SPORTS)
4.     {cout<<"\n Athlete
   constructor.."<<endl;}
5.   Athlete(int n , char *p
6.     ):Human(n,p),atcat(SPORTS)
7.     {cout<<"\n Athlete(int n, char *p)
   constructor.."<<endl;}
8.   ~Athlete(){}; // Default destructor
9.   BAT GetBAT() const { return atcat;}
10.  void SetBAT ( BAT c) { atcat=c;}
11.  // Athletes do differently functions
12.  void Play() { cout<<"\n Athletes
   play at college and at home... "<<endl;}
```

```
12. void UseGym(){ cout<<"\n Athletes
use GYM for advancement... "<<endl;}
13. private: BAT atcat; // Athletes
category
14. };
15. void main()
16. { Human hum(53,"Ramesh");
17. Student std(27,"Anand");
18. Athlete atd(25,"Gautam");
19. cout<<"\nHuman name is
:"<<hum.GetName()<<endl;
20. cout<<"\nStudents name is
:"<<std.GetName()<<endl;
21. cout<<"\n Athletes name is
:"<<atd.GetName()<<endl;
22. cout<<"\n Objectof Human..."<<endl;
23. hum.Work();
24. hum.Learn();
25. cout<<"\n Objectof Student..."<<endl;
26. cout<<"\n Student works like Human.
Student inherits Work() from
Human"<<endl;
27. std.Work();
28. cout<<"\n Student does his own thing
like UseNet "<<endl;
29. std.UseNet();
30. cout<<"\n Objectof Athlete..."<<endl;
31. cout<<"\n Athlete learns like Human.
Athlete inherits Learn() from
Human"<<endl;
32. atd.Learn();
33. cout<<"\nAthlete does his own thing
```

```

like UseGym "<<endl;
34. atd.UseGym();
35. }
/*Output :Human constructor..
Human constructor..
Student(int n, char *p) constructor..
Human constructor..
Athlete(int n, char *p) constructor..
Human name is :Ramesh : Students name is
:Anand : Athletes name
is :Gautam
Objectof Human. Human work . : Human
beings strive to learn :
Objectof Student.
Student works like Human. Student
inherits Work() from Human
Human work
Student does his own thing like UseNet
:Students use internet
for learning..
Objectof Athlete : Athlete learns like
Human. Athlete inherits
Learn() from Human
Human beings strive to learn
Athlete does his own thing like UseGym :
Athletes use GYM for
advancement
Human destructor.. Human destructor.. :
Human destructor..*/

```

---

We leave analysis of output statements and invoking of constructors and destructors vis à vis the code as an exercise for the reader.

## 12.6 Base Class Function Hiding

When a derived class overrides a base class function and if the base class function has been overloaded three times, then except for the function that is overridden, the other two overloaded functions are hidden and not available. For example, let us say that `Work()` has been overloaded in a base class called `Human` as shown below:

---

```
void Work()const { cout<<"\n Human work
..."<<endl;}
void Work(int n) const {cout<<"\n Human
work full speed..."<<endl;
void Work(int n, int p) const {
cout<<"\n Human work round the
clock..."<<endl;
```

---

Now if derived class `Student` overrides one of the base class say **`void Work()`**, then the balance two functions, `void Work(int n) const` and `void Work(int n , int p)`

`const`, are hidden and not available using derived class `Student` object.

### **Example 12.5: `inherit4.cpp`**

### **Program to Show Base Class Function Hiding**

```
1. #include<iostream>
2. using namespace std;
3. class Human
4. {public:
5. void Learn()const{ cout<<"\n Human
beings strive to learn    ..."<<endl;}
6. void Learn(int f)const{ cout<<"\n
Humans learn at good speed...    "<<endl;}
7. void Learn(int f, int g) const{
cout<<"\n Humans learn at good    speed.&
well"<<endl;}
8. void Work()const { cout<<"\n Human
work ..."<<endl;}
9. };
10. class Student:public Human
11. { public:
12. /* students do differently
functions. So it overrides base class
```



```

Learn() function. But base class has
Learn() as over loaded function.
Therefore balance two overloaded
functions are lost to Student.*/
13. void Learn(){ cout<<"\n Students
strive to learn professional
subjects..."<<endl;}
14. };
15. void main()
16. { Human hum; // object of base class
17. Student std; // object of derived
class
18. cout<<"\n Object of base class can
call all varieties of overloaded
function.."<<endl;
19. hum.Learn();
20. hum.Learn(2); // Learn with double
speed
21. hum.Learn(2,2); // Learn with good
speed and well
22. cout<<"\n Objectof Student..."<<endl;
23. cout<<"\n Student over rides Learn()
of base class"<<endl;
24. std.Learn(); //std over ride one of
the base class function Learn()
25. cout<<"\n Two other base class over
loaded functions Learn(int),
Learn(int,int) are lost"<<endl;
26. //std.Learn(2); //lost
27. cout<<"\nstd can still call an
overloaded base class function by
explicit call"<<endl;

```

```

28. std::Human::Learn(2);
}
/*Output : Object of base class can call
all varieties of overloaded function..
Human beings strive to learn : Humans
learn at good speed
Humans learn at good speed & well
Objectof Student. : Student overrides
Learn() of base class
Students strive to learn professional
subjects
Two other base class overloaded
functions Learn(int), Learn (int,int)
are lost
std can still call an overloaded base
class function by explicit call
Humans learn at good speed */

```

---

## 12.7 Virtual Functions

If function is declared as virtual function in base class, then we can execute an overriding function with the same name in the derived class with a pointer to base class. This means that we have

---

```

Inheritance relation available. Ex
Student is a Human
Pointer to base class must be declared
and memory obtained from

```

Heap.

Ex `Human * ptr = new Student;` This is a valid assignment since Student **is** a Human.

---

With a pointer to base class we can access all the member data and functions of derived class. But for this magic to work we have to declare the base class functions that are going to be overridden in derived class as virtual function like this

---

```
In base class
virtual void Work() const { cout<<"\n
Human work .....base class.";}

In derived class
void Work()const { cout<<"\n Students
work in
Laboratories.."<<endl;}
```

---

We can, in addition call base class functions which have been normally overridden and requires that an exclusive call has to be made like **`std::Human::Work()`** ; with a statement shown: **`cout<< ptr->GetRollNo()`** ; We

have shown all the above concepts are working in our next example:

## **Example 12.6: inherit5.cpp Virtual Functions**

```
//inherit5.cpp Virtual Functions :  
declare a base class called Human and a  
//derived classes called Student  
1. #include<iostream>  
2. using namespace std;  
3. enum CAT { BTECH,MBA,MCA,MSC}; //  
Category of student  
4. class Human  
5. {public:  
6. Human():idNo(0),name("Noname")  
7. {cout<<"\nHuman  
constructor.."<<endl;} //Base  
constructor  
8. ~Human(){cout<<"\nHuman  
destructor.."<<endl;} // base destructor  
9. // Public functions  
10. void Learn()const{ cout<<"\n Human  
beings strive to learn ..."  
<<endl;}  
11. virtual void Work()const { cout<<"\n
```

```

Human work ..."<<endl;}
12. protected: int idNo; char *name;
13. };
14. class Student:public Human
15. { public:
16. Student() {cout<<"\n Student
constructor..."<<endl;}
17. ~Student(){}; // Default destructor
18. // functions over ride base class
19. void Learn(){ cout<<"\n Students
work at college laboratories..."<<endl;}
20. // Students own function
21. void UseNet(){ cout<<"\n Students
use internet for advancement..."<<endl;}
22. // over riding of base class virtual
function
23. void Work(){ cout<<"\n Students
strive to learn professional
subjects\n";}
24. private:
25. // kept empty purposely to make
understanding of the program easy
26. };
27. void main()
28. {Human *pstd = new Student;
29. cout <<"\n Calling overriding
function Learn()..."<<endl;
30. cout<<"\nThough object is derived
class as the pointer is to base
class\n";
31. cout<<"\n you can see its base class
Learn() that is accessed"<<endl;

```

```

32. pstd->Learn();
33. cout <<"\ncalling a virtual function
with pointer to base class.."<<endl;
34. cout<<"\nThough pointer is to base
class , because its virtual
function"<<endl;
35. cout<<"\n you can access derived
class function.."<<endl;
36. pstd->Learn();
37. }

/*Output: Human constructor..
Student constructor.. Calling over
riding function Learn()
Though object is derived class as the
pointer is to base class
you can see its base class Learn() that
is accessed
Human beings strive to learn
calling a virtual function with pointer
to base class..
Though pointer is to base class,
because its virtual function
you can access derived class function..
Human beings strive to learn */

```

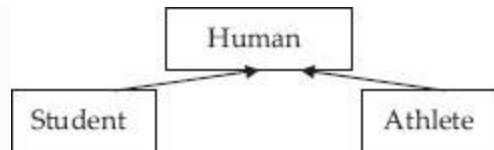
---

<b>Line No. 11:</b>	human declares a virtual function learn() to be overridden by derived classes
---------------------	---

<b>Line No. 23:</b>	Derived class overrides base class virtual function.
<b>Line No. 28:</b>	Human *pstd = new Student; The object is new student on the heap memory.
	But the pointer is to base class Human.

## 12.8 Multiple Virtual Functions

The concept of virtual function can be extended to cover multiple virtual functions. For example, base class can define a virtual function and there can be several derived classes with their own implementations. Depending on the implementation chosen by the user, virtual function pointer can invoke the particular implementation by the derived class. The situation diagram is shown in [Figure 12.3](#).



**Figure 12.3** Two derived classes

## **Example 12.7: inherit6.cpp**

### **Multiple Virtual Functions**

```
1. //Example 12.6 inherit6.cpp multiple
   virtual functions
2. //declare a base class called Human
   and two derived classes called Student
   and Athlete
3. #include<iostream>
4. using namespace std;
5. class Human
6. {
7. public:
8.     Human():idNo(0){} //Base
   constructor
9.     ~Human(){} // base destructor
```



```
10.    // Public functions
11.    virtual void Work()const {
cout<<"\n Human work
    ..."<<endl;}
12. protected:
13. int idNo;
14. };
15. class Student:public Human
16. {public:
17. // Own implementation of virtual
function Work()
18. void Work()const{ cout<<"\n Students
work at college labs.."<<endl;}
19. };
20. class Athlete:public Human
21. {public:
22. // Athletes Own implementation of
virtual function Work()
23. void Work()const{ cout<<"\nAthletes
workout at Gym after study hour\n";}
24. };
25. void main()
26. { Human *hum[2]; // array of
pointers to base class
27. Human *ptr; //pointer to base class
28. int i=0;
29. for ( i=0;i<3;i++)
30.     { if(i==1)
31.         ptr=new Student;
32.         else
33.         {if(i==2)
34.             ptr=new Athlete;
```

```

35.         else
36.             ptr=new Human;
37.         }
38.     hum[i]=ptr;
39. } // end of while
40. for ( i=0;i<3;i++)
41.     hum[i]->Work();
42. }

/*Output Human work ...
Students work at college laboratories...
Athletes workout at college Gym after
study hours...*/

```

<b>Line No. 26:</b>	Human *hum[2]; // array of pointers to base class.
<b>Line No. 27:</b>	Is a pointer to base class.
<b>Line No. 31, 34 &amp; 36:</b>	Produces object of new Student, or new Athlet, or new Human depending on if else statements
<b>Line No. 38:</b>	allocates pointer to hum[i] and Line No. 40 & 41 run a loop to display hum[i]->work()

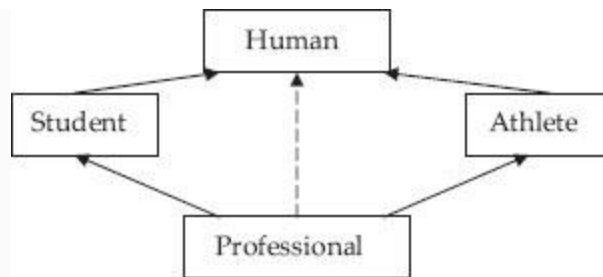
---

## 12.9 Virtual Destructors: Why and How?

In inheritance relation, we know that a Student **is** Human. Therefore, we can pass a pointer to derived class instead of pointer to base class. Now when we try to delete the object, the derived class destructor should be called first, which in turn calls the base class destructor. For this to happen, the base class destructor must be declared as virtual. Note that the constructor **cannot be** declared as virtual.

## 12.10 Hybrid Inheritance with Multiple Inheritances

In some situations, single inheritance is not sufficient and leads to ambiguity. Consider the example presented in Figure 12.4.



**Figure 12.4** Hybrid inheritance

A Student works at college laboratories and an Athlete works out at the Gym after college hours. But let us say the recruiting companies need a Professional who works at collage laboratories and also takes keen interest in Gym and sports. We need a professional who inherits `Work()` of Students and also `Work()` of Athlete on requirement. In addition, the Professional must retain all the basic qualities of Human (shown as dotted arrow).

**Multiple inheritance is deriving a class from more than one base class.** Also notice in the figure that both Student and Athlete have the same base class. This type of base class is **called virtual base**

**class** because though there is only one base class common to both derived classes, each of the derived classes feels it has its own base class.

## 12.11 Virtual Inheritance

Refer to Figure 12.4 once again. The Professional is deriving properties and qualities both from Student and Athlete depending on the need. In order to facilitate inheritance of properties, functions and member data selectively from Student and Athlete, we need to declare these two classes as **virtual inheritance from the base class**. This concept is illustrated in our next example. The example we have chosen is that companies want to select professionals who are good students and good athletes and good humans. Accordingly, we have introduced fields and variables

---

```
enum OKAY{FALSE,TRUE}; // eligible for
selection or not.
enum DIV { DISTINCTION,FIRST, SECOND,
NATIONAL, STATE, COLLEGE};
for recoding achievements of athletes.
```

Num or n no of companies willing to give weightage to all round development

---

## **Example 12.8: inherit7.cpp Virtual Inheritance**

---

```
1. #include<iostream>
2. using namespace std;
3. enum CAT { BTECH,MBA,MCA,MSC}; //
Category of student
4. enum BAT { ATHLETE,SPORTS};
5. enum DIV { DISTINCTION,FIRST,SECOND
,NATIONAL, STATE, COLLEGE};
6. // Division obtained by Students
7. enum OKAY{FALSE,TRUE}; // eligible
for selection or not
8. class Humans
9. {public:
10. Humans(int n):idNo(n)
11. {cout<<"\n default Humans
constructor.."<<endl;} //Base
constructor
12. virtual ~Humans(){cout<<"\nHumans
destructor.."<<endl;} //
```

```

base destructor
13. // Public functions
14. virtual int GetId()const{return
idNo;}
15. virtual void SetId(int n) { idNo=n;}
16. protected:
17. int idNo;
18. };
19. class Student:virtual public Humans
20. { public:
21. Student(CAT cat , DIV d , int num);
22. virtual ~Student(){}; // Default
destructor
23. // students do differently functions
24. virtual void Learn()const{ cout<<"\n
Students learn relevant courses\n";}
25. virtual void Work()const{ cout<<"\n
Students worked Industry programs\n";}
26. virtual CAT GetCAT() const { return
stcat;}
27. virtual void SetCAT(CAT c) {
stcat=c;}
28. virtual DIV GetDIV() const { return
division;}
29. virtual void SetDIV ( DIV d) {
division = d;}
30. protected:
31.     CAT stcat; // students category
32.     DIV division; // division
Distinction/first class etc
        obtained
33. };

```

```

34. Student::Student(CAT cat , DIV d ,
int idnum):
35.     Humans(idnum),stcat(cat),
division(d)
36.     { cout<<"\n Students
constructor.."<<endl;}
37. class Athlet:virtual public Humans
38. { public:
39. Athlet(CAT cat , DIV d , OKAY ok ,
int idnum);
40. virtual ~Athlet(){}; // Default
destructor
41. virtual void AtLearn()const{
cout<<"\n Athlets learn subjects &
sports\n";}
42. virtual void AtWork()const{
cout<<"\n Athlets worked at lab and
GYM\n";}
43. virtual void Play() const{ cout<<"\n
Athlets can play games and events\n";}
44. virtual CAT GetCAT() const { return
stcat;}
45. virtual void SetCAT(CAT c) {
stcat=c;}
46. virtual DIV GetDIV() const { return
division;}
47. virtual void SetDIV ( DIV d) {
division = d;}
48. virtual OKAY GetOKAY() const {
return qualified;}
49. virtual void SetOKAY ( OKAY q) {
qualified = q;}

```



```

50. private:
51.     CAT stcat;
52.     DIV division; //
national/state/district/college
53.     OKAY qualified;
54. };
55. Athlet::Athlet(CAT cat , DIV d ,
OKAY ok, int idnum):
56.
Humans(idnum),stcat(cat),division(d),qua
lified(ok)
57. {cout<<"\n Athlet
constructor.."<<endl;}
58. class Professional : public Student
, public Athlet
59. { public:
60. Professional(CAT cat , DIV d, OKAY q
,int idnum , int n); ~Professional()
{cout<<"\n Professionals
destructor.."<<endl;}
61. // n is no of companies giving
weightage to all round development
62. virtual GetNum() const { return
num;}
63. //num : these are companies willing
to give weightage to sports
64. virtual DIV GetDIV() const { return
Student::division;}
65. void AtLearn() const { Learn();} //
Learn subjects like students
66. void AtWork() const { Work();} //
Work in Laboratories like Students

```

```

67. private:
68. int num ; //Number of companies who
are willing to give weightage to sports
69. };
70. Professional::Professional(CAT cat ,
DIV d, OKAY q ,int idnum, int n):
71.
Student(cat,d,num),Athlet(cat,d,q,idnum)
,Humans(idnum),num(n)
72. { cout<<"\n Professionals
constructor.."<<endl;}
73. void main()
74. {Professional *ptr = new
Professional(BTECH,FIRST,TRUE,50595,100)
;
75. int rollNo = ptr->GetId();
76. ptr->Learn();
77. //Though we call AtLearn() of Athlet
, it will call learn() of Student.
78. //We want student to be sound in
Knowledge.
79. ptr->AtLearn();
80. ptr->Play(); // play like an athlete
81. cout<<"\n No of companies allocating
weightage to all round personality : "
<<ptr->GetNum()<<endl;
82. cout<<"\n Division obtained by the
professional : "<<ptr->GetDIV()<<endl;
83. cout<<"\n The selected Professional
is : " <<rollNo<<endl;
84. delete ptr;
85. }

```

```

/*Output : default Humans constructor..
Students constructor.. : Athlet
constructor...: Professionals
constructor..
Students learnt Industry relevant
courses: Students learnt Industry
relevant courses
Athletes can play games and participate
in events
No. of companies allocating weightage to
all round personality: 100
Division obtained by the professional: 1
Companies tion The selected Professional
is: 50595
Professionals destructor..
Humans destructor*/

```

---

<b>L i n e N o . 1 9 &amp; 3</b>	are statements showing Student and Athlete using virtual inheritance form single base class Human.
--	--

7  
:

**L** Student declares virtual functions Learn() and  
**i** Work() so that they can be inherited by  
**n** Professional if he needs them. Similarly, Line  
**e** No. 41 to 49 define virtual functions for Athlete  
**N** class.

**o**  
**.**  
**2**  
**4**  
**&**  
**2**  
**9**  
**:**

**L** are virtual destructors.

**i**  
**n**  
**e**  
**N**  
**o**  
**.**  
**1**  
**2**  
**,**  
**2**  
**2**  
**,**  
**&**  
**4**

**o**  
**:**

**L**  
**i**  
**n**  
**e**  
**N**  
**o**  
**.**  
**7**  
**o**  
**&**  
**7**  
**1**  
**:**

Professional::Professional(CAT cat,  
DIV d, OKAY q, int idnum, int n)  
Student(cat,d,num) ,  
Athlet(cat,d,q,idnum) ,  
Humans(idnum) , num(n) are for constructor  
for Professional. Observe that Student, Athlet  
and Human constructors are called in turn to  
create an object called Professional

**L**  
**i**  
**n**  
**e**  
**N**  
**o**  
**.**  
**7**  
**6**  
**:**

ptr->AtLearn() ; Though we call  
AtLearn() of Athlet , it will call learn() of  
Student. We want student to be sound in  
Knowledge.

## 12.12 Run-time Polymorphism and Dynamic Binding

Inheritance solves the problem of reusability. That is derived class can access all the data members and function members of base class that are declared as protected. We have also learnt that derived class can override the functions defined in base class. Further, we have seen while dealing with virtual functions that with a pointer to base class we can call the derived class object

---

```
Human *ptr=new Student;
```

---

Combining the above two features C++ gives us a powerful tool, called run-time polymorphism and dynamic binding. What this feature means to the programmer is that with a pointer to base class, we can decide at run-time to call a particular derived classes overriding function. In other words, we can bind the overriding function from several of the derived classes, with pointer to base class.

## 12.13 Abstract Data Types (ADTs)

The features described above, namely, run-time polymorphism and dynamic binding will allow us to define a base class with virtual functions with no implementation (called pure virtual functions) or with dummy functionality just to indicate to the user that implementation is by derived class. These classes are called abstract data types because they hide the implementation.

We will declare a base class called Shape. You will appreciate Shape has no definite shape, no defined area, perimeter or specific draw routine. We will derive classes like Circle and Square and provide the **solid** implementation of these virtual functions. Solid means all virtual functions will be implemented in each of the derived class. The concepts discussed are at work in our next example.

**Example 12.9:      inherit8.cpp**  
**ADT – Run-time Polymorphism and**

# Dynamic Binding

```
1. #include<iostream>
2. using namespace std;
3. const double PI=3.14159;
4. class Shape
5. { public:
6. Shape(){};
7. ~Shape(){}
8. virtual double ComputeArea()
9. {cout<<"\n Shape has no definite Area
"<<endl;return 0.0;}
10. virtual double ComputePerimeter()
11. {cout<<"\n Shape has no definite
Area "<<endl; return 0.0;}
12. virtual void DrawShape() {} };
13. class Circle: public Shape
14. {public: Circle(double r):radius(r)
{};
15. ~Circle(){}; // Default destructor
16. // Circles do functions differently
17. double ComputeArea()
18. {cout<<"\n Circle implementation of
ComputeArea "<<endl;
19. return PI*radius*radius; }
20. virtual double ComputePerimeter()
21. {cout<<"\n Circle implementation of
ComputePerimeter <<endl;
22. return 2*PI*radius; }
```



```

23. virtual void DrawShape() {cout<<"\n
Circle Draw code here" <<endl;}
24. private:double radius;
25. };
26. class Square: public Shape
27. {public: Square(double r):side(r){};
28. ~Square(){}; // Default destructor
29. // Squares do functions differently
30. double ComputeArea()
31. {cout<<"\n Square implementation of
ComputeArea " <<endl;
32. return side*side; }
33. virtual double ComputePerimeter()
34. {cout<<"\n Square implementation of
ComputePerimeter " <<endl;
35. return 4*side; }
36. virtual void DrawShape() {cout<<"\n
Square Draw code here"<<endl;}
37. private:double side;
38. };
39. void main()
40. { int choice;
41.     Shape *ptr; // pointer to base
class
42.     while(choice!=3)
43.     { cout<<"\n 1 : Circle 2 : Square
3: Exit .. :";
44.     cin>>choice;
45.     switch(choice)
46.     { case 1 : ptr = new
Circle(2.0); break;
47.     case 2 : ptr= new Square(5.0);

```

```

break;
48.         case 3 : exit(0);
49.         default: cout<<"\n wrong
choice. Try Again.."<<endl; continue;
50.         }
51.         // Function Polymorphism and
Dynamic Binding at work
52.         ptr->ComputeArea(); // call to
derived class function
53.         ptr->ComputePerimeter();ptr->
DrawShape();
54.         }//end of while
55.         }//end of main
/*Output 1 : Circle 2 : Square 3: Exit
.. :1
Circle implementation of ComputeArea
Circle implementation of
ComputePerimeter
Cirle Draw code here
1 : Circle 2 : Square 3: Exit .. :2
Square implementation of ComputeArea
Square implementation of
ComputePerimeter
Square Draw code here
1 : Circle 2 : Square 3: Exit .. :3
*//*output :1 : Circle 2 : Square 3:
Exit .. :1
Circle implementation of ComputeArea
Circle implementation of
ComputePerimeter
Cirle Draw code here
1 : Circle 2 : Square 3: Exit .. :2

```

```
Square implementation of ComputeArea
Square implementation of
ComputePerimeter
Square Draw code here
1 : Circle 2 : Square 3: Exit .. :4
wrong choice. Try Again..
1 : Circle 2 : Square 3: Exit .. :3 */
```

---

## 12.14 Pure Virtual Functions

Sometimes we will need situations wherein base class should not have any solid implementation. Like we did not achieve anything in `ComputeArea()` or `DrawShape()` functions in base class. They are simply tools to create pointer to derived class functions wherein all virtual functions are implemented. In such cases, it is better to use pure virtual functions.

---

```
virtual double ComputeArea()=0;
virtual void DrawShape()=0;
```

---

By declaring a function in base class, we are telling the compiler that implementations are in derived class. Note that if a base class has pure virtual

functions, we cannot create an object of base class at all. Also all pure virtual functions must be implemented in derived class.

But if you have some common functionality which every derived class must implement, then repeating the code at each of the derived classes defeats the purpose of reusability. Thankfully, you can include such common code in the base class Shape and, for solid implementations, use derived class DrawShape() and on completion revert to base class for undertaking common functionality. We provide implementation of pure virtual function in the example that follows:

### **Example 12.10: inherit9.cpp ADT – Run-time Polymorphism and Dynamic Binding**

---

```
1. #include<iostream>
2. using namespace std;
3. const double PI=3.14159;
```

```

4. class Shape
5. {public: Shape(){};
6.     ~Shape(){}
7.     virtual double ComputeArea()=0;
//pure virtual function
8.     virtual double ComputePerimeter()
=0;
9.     virtual void DrawShape()=0;
10. };
11. // We will implement DrawShape (),
though it is a pure virtual Function
12. // to include common functionality
of all the derived functions
13. void Shape::DrawShape()
14. {cout<<"\n Completion of DrawShape()
mechanism from derived class..."<<endl; }
15. class Circle: public Shape
16. {public: Circle(double r):radius(r)
{};
17.     ~Circle(){}; // Default
destructor
18.     // Circles do functions
differently
19.     double ComputeArea()
20.     {cout<<"\n Circle implementation
of ComputeArea "<<endl;
21.     return PI*radius*radius;}
22.     double ComputePerimeter()
23.     {cout<<"\n Circle implementation
of ComputePerimeter "<<endl;
24.     return 2*PI*radius;}
25.     void DrawShape()

```

```
26.      {cout<<"\n Circle Draw code here"
<<endl;
27.      Shape::DrawShape();} // call to
base class to finish common
functionality
28. private: double radius;
29. };
30. class Square: public Shape
31. {public: Square(double r):side(r){};
32.      ~Square(){}; // Default
destructor
33.      // Square implements functions
differently
34.      double ComputeArea()
35.      {cout<<"\n Square implementation
of ComputeArea "<<endl;
36.      return side*side;}
37.      double ComputePerimeter()
38.      {cout<<"\n Square implementation
of ComputePerimeter "<<endl;
39.      return 4*side;}
40.      void DrawShape()
41.      {cout<<"\n Square Draw code here"
<<endl;
42.      Shape::DrawShape(); } // call to
base class to finish common
functionality
43. private:double side;
44. };
45. void main()
46. {int choice;
47. Shape *ptr; // pointer to base class
```

```

48. while(choice!=3)
49. {cout<<"\n 1 : Circle 2 : Square 3:
Exit .. :";
50.     cin>>choice;
51.     switch(choice)
52.     { case 1 : ptr = new Circle(2.0);
break;
53.         case 2 : ptr= new Square(5.0);
break;
54.         case 3 : exit(0);
55.         default: cout<<"\n wrong choice.
Try Again.."<<endl; continue;
56.     }
57.     ptr->ComputeArea(); ptr-
>ComputePerimeter(); ptr->DrawShape();
58. }//end of while
59. }

/*Output 1 : Circle 2 : Square 3: Exit
.. :1
Circle implementation of ComputeArea
Circle implementation of
ComputePerimeter
Circle Draw code here
Completion of DrawShape() mechanism from
derived class...
1 : Circle 2 : Square 3: Exit .. :2
Square implementation of ComputeArea
Square implementation of
ComputePerimeter
Square Draw code here
Completion of DrawShape() mechanism from

```

derived class...

1 : Circle 2 : Square 3: Exit .. :3

---

## 12.15 Summary

1. Reusability is the main reason why inheritance has been included as an important concept of OOP language.
2. Inheritance specifies is type of relation.
3. A derived class can inherit public or private or protected.
4. If inheritance is public then public and protected members of base class remain public and protected in derived class.
5. If inheritance is protected, the public and protected members become protected members.
6. If inheritance is private, the public and protected members become private members.
7. Single inheritance is when a derived class inherits only from one base class.
8. Multiple inheritance is when a derived class inherits from multiple base classes.
9. Hierarchical inheritance is when many derived classes inherit from a single base class.
10. Multilevel inheritance is when a derived class inherits from a class which has been inherited from another class.
11. Hybrid inheritance is when a derived classes inherits from multiple base classes and they in turn are derived from a single base class.
12. In hybrid inheritance, ambiguity may arise and to resolve the ambiguity use explicit call with scope resolution operator :: or declare the base classes as virtual. This is called virtual inheritance.



13. When a derived class overrides a base class function and if the base class function has been overloaded three times, then except for the function that is overridden, the other two overloaded functions are hidden and not available.
14. If the function is declared as virtual function in base class, then we can execute an overriding function with the same name in the derived class with a pointer to the base class.
15. In multiple virtual functions, the base class defines a virtual function and there can be several derived classes with their own implementations.
16. Destructors can be virtual but constructors cannot be virtual.
17. Run-time polymorphism means that we can bind the overriding function from several of the derived classes, with pointer to base class.
18. Classes are called abstract data types because they hide the implementation. A base class with virtual functions with no implementation (called pure virtual functions) or with dummy functionality indicates to the user that implementation is by derived class.
19. A pure virtual function means a base class should not have any solid implementation, but contains only names of the functions. For example: **virtual void DrawShape ()=0 ;** Pure virtual functions are simply tools to create pointer to derived class functions.

## Exercise Questions

### Objective Questions

1. Inheritance specifies ----- type of relation.

1. has
2. was
3. is
4. segregation

2. In public inheritance, public implies derived class can inherit

1. Only public variable
2. Only public functions
3. Both member functions and member data
4. (d) None of member functions and member data

3. If a function has an object it has access to

1. All public member functions and data
2. All private member functions and data
3. All protected members and data from objects from which they derive

1. i
2. i and ii
3. ii and iii
4. i, ii and iii

4. Public function of a derived class can access all the protected data of base class TRUE/FALSE

5. When you create the object for the derived class, the constructor for derived class is called first. TRUE/FALSE

6. When an object belonging to derived class has to be removed, the base class destructor will be called first. TRUE/FALSE

7. When a derived class overrides a base class overloaded function, the object of derived class can get access to

1. Only overridden function of derived class.
2. All the overloaded functions of base class.
3. All the overridden functions of derived class.

1. i only
2. ii only
3. i and ii
4. i and iii

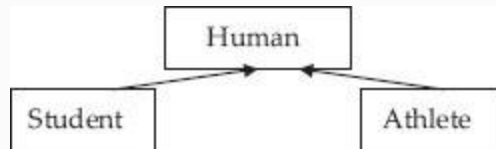
8. In C++ pointer to base class can be assigned to derived class. TRUE/FALSE

9. For run-time polymorphism and dynamic binding to work the following are required: Pointer to base class

1. Inheritance relation
2. Function to be declared as virtual
3. Function to be overridden in derived class

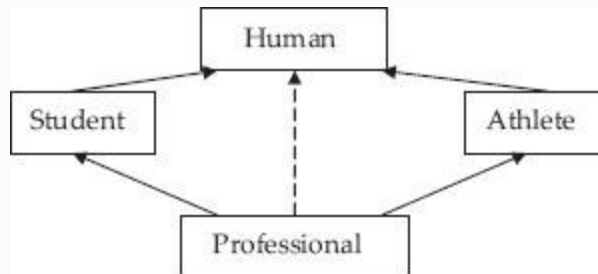
1. i and ii
2. i, ii and iii
3. i, ii, iii and iv
4. NOT i, ii, iii and iv

10. What is the name of the relation depicted in the picture?



1. Inheritance
2. Virtual base class
3. Multiple inheritance
4. All of a, b, c

11. What is the name of the relation depicted in the picture?



1. Virtual base class
2. Multiple inheritance
3. Inheritance
4. b and c

12. Pure virtual functions declaration in base class means

1. No implementation in base class only name
2. Common to all derived classes implementation can be included in base class
3. Derived classes must implement all the pure virtual functions.

1. Not i, ii and iii
2. i, NOT ii and iii
3. i, ii and iii
4. i, ii and NOT iii

### 13. Pure virtual function means

1. Object to class that has pure virtual function cannot be declared.
2. Derived class must define all pure virtual function in its class.
3. Virtual r/t function name() {};
4. Virtual r/t function name ()=0;
5. Pure virtual r/t function name ()=0;

1. i, ii, not iii, iv and not v
2. i, ii and iii
3. i, ii and v
4. i, ii and iv

### 14. Destructor need not be virtual if a class has virtual functions TRUE/FALSE

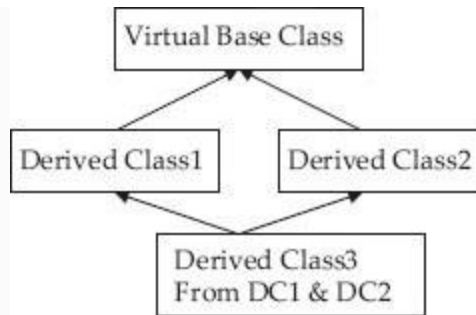
### 15. Constructor of class having virtual function can be virtual TRUE/FALSE

### 16. ADT are so-called because

1. They hide implementation
2. They bind member functions of derived class dynamically at run-time.
3. They have abstract data

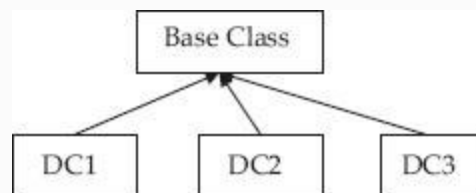
1. i
2. ii
3. iii
4. i and ii

### 17. What is the name of the relation depicted in the picture?



1. Hybrid inheritance
2. Virtual base class
3. Multiple inheritance
4. All of a, b and c

18. What is the name of the relation depicted in the picture?



1. Hierarchical inheritance
2. Virtual base class
3. Multiple inheritance
4. a, b and c

19. Protected members in a base class can be accessed

1. Only in derived class
2. Only in base class
3. Both i and ii (iv) None

20. In public inheritance, public members become public in derived class. TRUE/FALSE

21. In private inheritance, the inherited members can further be passed on to derived classe by class. TRUE/FALSE

22. In protected inheritance, public members & protected members retain their visibility in derived class. TRUE/FALSE

### **Short-answer Questions**

23. What is public inheritance? What is private inheritance? On what occasions do you use public or private inheritance?
24. What are the types of inheritance?
25. What is virtual inheritance?
26. Can constructors be virtual? Why or why not?
27. Can destructors be virtual? Why or why not?
28. Explain base class overriding.
29. Explain multiple virtual functions.

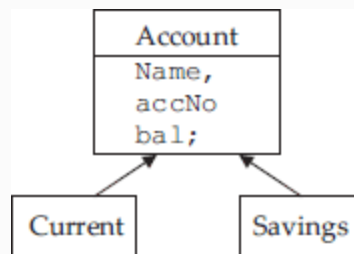
### **Long-answer Questions**

30. Discuss the access specifiers and the visibility in inheritance relationship.
31. Compare `is a` and `has a` relationship with suitable examples.
32. Distinguish between overloading and base class overriding. What is base class function hiding?
33. What are virtual functions? How are they different from pure virtual functions?
34. Explain run-time polymorphism and dynamic binding.
35. What are abstract classes? How are they useful in achieving reusability in OOPs?
36. Explain how hybrid inheritance leads to ambiguity and this ambiguity can be resolved.

### **Assignment Questions**

37. What does inheritance mean in C++? What are the different types of inheritances? Give examples for each type.
38. What is a virtual function? Why do we need virtual functions? What are the rules for virtual function implementation?
39. What is an abstract class? Why do we need abstract classes? What are the rules governing virtual base class? Implement a program covering the above aspects.

40. Are friendship and inheritance contradicting in their roles? When do you use friend functions and when do you use inheritance.
41. Containment and inheritance have been introduced to achieve reusability. Bring out the similarities and differences through examples.
42. Are ADT and pure virtual functions one and the same or are they different? Explain with suitable examples.
43. Bring out the differences in constructor invocation in case of nested classes and inherited classes.
44. The relationships in a bank are shown below. The class diagram shown also indicates the functions and member data. The additional rules are Type of account : current and savings  
Minimum Balance: Rs 5000/– for current and Rs 1000/– for savings.  
Penalty: Rs 100/– if the balance falls below and also makes the account inactive.



The functionality of banking system to include  
Update transaction of deposited amount and display  
Transaction details.  
Compute annual interest compounded twice at 10% per  
annum of interest.

## Solutions to Objective Questions

1. c

2. c
3. d
4. True
5. False
6. False
7. d
8. True
9. c
10. c
11. b
12. c
13. a
14. False
15. False
16. d
17. d
18. a
19. c
20. True
21. False
22. True



# 13

## IO Streaming

### LEARNING OBJECTIVES

*At the end of the chapter, you will be able to understand and program*

- Unformatted and formatted stream operators.
- Use IO manipulator to format output.
- Sequential and random/direct access file handling.
- Read objects from and write objects onto a file

### 13.1 Introduction

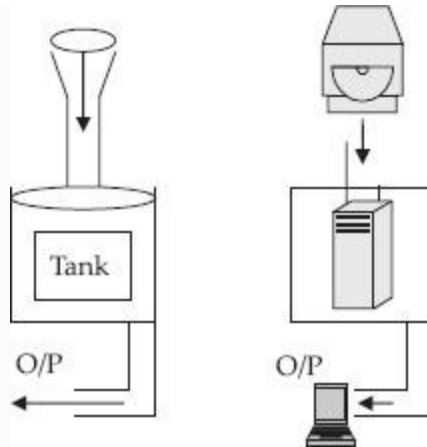
What crosses your mind when you think of streams? Probably streams of water, we call them springs, flowing from the top of a hill

to plains. In C++, we call streams or IO streams.

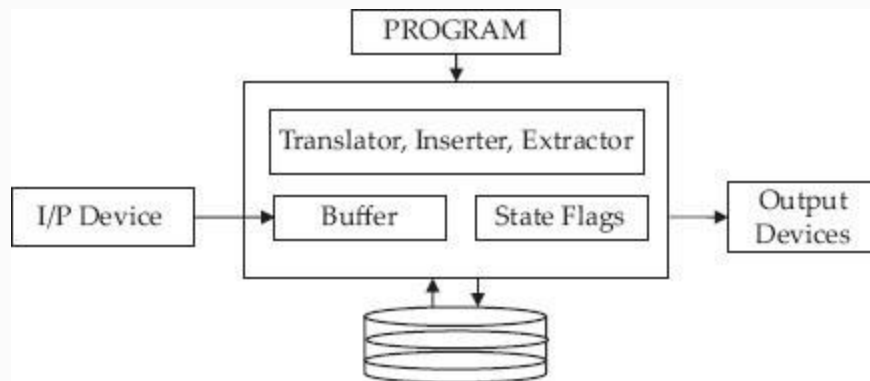
This chapter will introduce to you unformatted and formatted input and output streams. It deals extensively with stream manipulators that are used to format and control stream data. The chapter also concentrates on file handling in C++ through `<fstream>` header file. The concepts covered are sequential file handling and random/direct access techniques. The techniques to handle formatted data and raw data are discussed. We will wrap up the chapter with object read and write operations onto a file.

## 13.2 IO Streaming

In C++, IO streams are flow of bytes from one input device to memory and vice versa. A stream is like a pipe. It can carry anything in the pipe, be it kerosene, milk, water, etc. – refer to Figure 13.1 shown below.



**Figure 13.1** Streams – A concept



**Figure 13.2** IO Streaming – A close look

Streams output whatever is inputted.

- **Byte Stream** **int in:** int out
- **string** **in:** string out

- **object in:** object out

### 13.3 IO Library Files Classification

The library functions are classified as Console IO, Disk IO and Port IO. Port IO are used for input and output programming, when we want to use ports for data input and output.

**Disk IO:** This mode of operation is performed on entities called files. Usually, writing onto files and reading from the files are never done directly onto disk. Instead, a buffer (a memory) is used to store prior to writing onto file and after reading from the file. Buffering in case of Disk IO is essential for saving access time required to access memory. DiskIO is of two types:

- High-level disk IO also called **standard IO/Stream IO:** Buffer management is done by the compiler/operating system.
- Low-level disk IO also called **system IO:** Buffer management is to be taken care of by the programmer.

**Console IO:** All the input and output functions control the way input has to be fed

and the way output looks on standard output devices, i.e. screen. These IO statements are further classified as formatted and unformatted.

- **Formatted IO:** We use cin and cout objects. We specify the format formatting commands of C++.
- **Unformatted IO:** The unformatted category for input and output.

### *13.3.1 Formatted IO*

In order to perform I/O operations, a stream is attached to an I/O device. Typical I/O devices include consoles, keyboards, files and electronic devices like sensors. The inserters represented by the "<<" symbol work as translators which translate in memory representation of data types such as integers, floats and user-defined data types into data types which can be understood by the I/O devices. The extractor represented by the ">>" symbol translates data sent by I/O devices to in memory representation formats such as integers and floats. Typical I/O devices like consoles and keyboards send and receive data in ASCII format.

Let us see with an example how extractors and inserters work. Consider the following piece of code `int n; cin >> n;` Now using the keyboard, assume the user has entered numbers 6, 7 and 8 followed by the enter key. Following this operation, the buffer associated with the input stream would look like Figure 13.3.



**Figure 13.3** Representations of input stream

ASCII representation of individual numbers is as follows:

---


$$\begin{array}{lcl}
 6 = 0 \times 36 = 0011 \ 0110 & 7 = 0 \times & \\
 37 = 0011 \ 0111 & 8 = 0 \times 38 = 0011 \ 1000 & 
 \end{array}$$


---

Through the statement `cin >> n` extractor knows that the input stream should be formatted into an integer as `n` is of the type integer. The extractor then proceeds to

extract each character from the stream and converts it into decimal value. To get equivalent decimal values from ASCII format, the extractor subtracts ASCII value of 0(0 × 30) from each character in the stream.

---

$$\begin{aligned} & \text{'6'} - \text{'0'} = 0 \times 36 - 0 \times 30 = 6 : \text{'7'} \\ & - \text{'0'} = 0 \times 37 - 0 \times 30 = 7 : \text{'8'} - \text{'0'} \\ & = 0 \times 38 - 0 \times 30 = 8 \end{aligned}$$

---

The extractor continues to do the above operation till it encounters '\n', carriage return (CR) in the input stream. It also stops extracting if it encounters Tab, Space and EOF markers. During extraction, the extraction operator also keeps a count of numbers or characters it has extracted and accordingly sets the place value of each digit. In our case, 6 is assigned a place value of 100, 7 a place value of 10, and 8 a place value of 1. The last step involves multiplying the extracted decimal values with their corresponding place values and adding them up to get integer representation:  $n = 6*100 + 7*10 + 8*1 = 678$ . But in memory,

representation is in the form of integer, so 678 will be represented as:

---

$$678 = 0 \times 2A6 = 0000 \ 0010 \ 1010 \ 0110.$$

---

The inserter works in exactly the opposite way. As seen from the above example, `iostream` library object `cin` coupled with the extractor operator `>>` provides the user translation services and also the processing of special characters such as Carriage Return (CR) , Space, Tabs and End of File (EOF) . This kind of I/O is called formatted or high-level I/O. Formatted I/O abstracts the user from inner workings of streams and is easier to use. The disadvantage of formatted I/O is that it is slow and not suitable for large data sizes.

### *13.3.2 Unformatted IO*

Unformatted I/O is also called low-level I/O. It offers the highest efficiency and throughput amongst all the other I/O mechanisms. Unlike formatted I/O, here



input has no automatic whitespace, Tab, carriage return detection and processing, and also data is not formatted – the programmer has to interpret the data.

## **Input:**

### **get() :**

- Reads a character from stream including delimiters like white spaces, EOF markers and returns it.
- Usage `cin.get()`

### **get(char& ch) :**

- Reads a character from stream and stores it in ch
- Usage `cin.get()`

### **get(char\* str, int count, char delim=' \n' ) :**

- Function reads count-1 characters or till it encounters the delimiter character in the stream and stores the characters in the str buffer.
- When the delimiter is encountered, it is not copied into the str buffer. It remains in the input stream
- Null character is inserted in the array str
- delimiter has to be flushed from stream else it will remain in the stream
- Usage `cin.get(buffer, count) //when you want to use default delimiter ' \n'`

- Usage `cin.get(buffer, count, '#')` //when you want to use different delimiter

**`getline(char* str, int count, char delim='\n') :`**

- Operates like `get(char* str, int count, char delim= '\n')`:
- When the delimiter is encountered, it is not copied into the str buffer but unlike `get()`, `getline()` discards delimiter from stream
- Usage: similar to `get(char* str, int count, char delim='\n')`

**`read(char* str, int count) :`**

- Function reads count characters from the stream and place in array str.
- Unlike the formatted I/O functions read does not append string termination character to the buffer str.
- If EOF marker is encountered during read operation function returns error.

## **Output:**

**`put(char ch) :`**

- Function writes the character ch to the stream.
- Usage: `cout.put(ch) ;`

**`write(const char* str, int count) :`**

- Functions writes count characters from the buffer str to the stream.
- Write operation does not terminate for any of the delimiters.
- Precautions should be taken to ensure count number of characters are present in the buffer str.
- Usage: `cout.write(str, count)`

## Miscellaneous:

### **putback(char c)**

- Places the character c obtained by `get()` back in to the stream.

### **peek()**

- Returns the next character from the stream without removing it

### **ignore( streamsize num=1, int delim=EOF ):**

- Skips over a designated number of characters (default is 1)
- Terminates upon encountering a designated delimiter (default is EOF)

Consider the following piece of code:

---

```
cout <<"\n Enter source file name...:
";
cin>>filename;
```

```
cout<<"\n Enter text for inputting to  
file ..."<<endl;  
cin.ignore(1, '\n');  
cin.getline(text, 80);
```

---

Now suppose you have entered `thunder.txt` and pressed enter at the first prompt the operation `cin>>filename` reads the input stream stores filename with the value `thunder.txt` but the `cin` operator does not extract the extra enter (`'\n'`) key that you have inputted. Now suppose that we do not include `cin.ignore(1, '\n')` and directly call `cin.getline(text, 80)`, it will return `NULL` as first character in the stream is `'\n'` the default delimiter for `getline()`. To avert this situation, we ignore the operation to discard the `'\n'` character present in the input stream.

**Example 13.1: `unformattedio1.cpp`  
Program to Show `cin.get()` to**

## Read a Character

```
1.  #include<iostream>
2.  #include<cstring>
3.  using namespace std;
4.  void main()
5.  {char ch, str[10];
6.   cout<<"\n Enter any string : Showing
character input thru cin.get() :";
7.   while((ch = cin.get()) != '\n') //
using get() function
8.   cout << ch;
9.   cout<<"\n Enter a string<Less Than 9
or less characters : cin.read>: ";
//using read fn
10.  cin.read(str, sizeof(str));
11.  str[9] = '\0';
12.  cout << endl << str;
13. }
```

/\*Output: Enter any string: Showing  
character input through cin.get() :Hello  
World  
Hello World  
Enter a string<Less Than 9 or less  
characters : cin.read>: hello USA\*  
Line No 7 Shown cin.get() . Line No 10  
shows cin.read() . Note that read() also  
works with cin object.

## Example 13.2: unformatio2.cpp

### Program to Show Unformatted Output Functions

```
1.  hello USA*/
2.  #include<iostream>
3.  #include<conio.h>
4.  #include<string.h>
5.  using namespace std;
6.  void main()
7.  { int i;
8.  char str[] = "I enjoy programming in
C++" ;
9.  for(i = 0 ; str[i] != '\0'; i++)
cout.put(str[i]); //using write function
10. cout<<endl; cout.write(str,
strlen(str)); getch();
11. }
/*Output I enjoy programming in C++ I
enjoy programming in C++*/
Line No 9 shows usage of cout.write().
Note that write works with cout also.
```

### 13.3.3 IO Stream State

I/O library in C++ provides a mechanism to test whether a particular I/O operation has succeeded or failed. Each stream object maintains a set of flags which indicate the state of the stream after an I/O operation. Library also provides set of member functions which can be used to test the flags. The flags and the corresponding member functions are shown in the Table 13.1.

**Table 13.1** Flags and member functions

Flag Name	Description	Testing Function
-----------	-------------	------------------

good bit	Set to 1 when stream encounters no errors	good()  Ex: cin.good() 
eofb it	Set to 1 when stream encounters end of file	eof()  Ex: cin.eof() 
fail bit	Set to 1 when stream encounters an error but is recoverable	fail()  Ex: cin.fail() 
badb it	Set to 1 when stream is corrupted and is unrecoverable	bad()  Ex:cin.bad() 



## Example 13.3: `streamstate.cpp` Program to Show Reading Stream State Flags

```
#include<iostream>
#include<conio.h>
using namespace std;
void main()
{int value;
  cout << "Enter Value: ";cin >> value;
  cout <<endl <<"goodbit: " <<
  cin.good();cout <<endl <<"eofbit : " <<
  cin.eof();
  cout <<endl <<"failbit: " <<
  cin.fail();cout <<endl <<"badbit : " <<
  cin.bad();
  cout <<endl <<"Enter Value (enter
  characters to see effect on flags):
  ";cin >> value;
  cout <<endl <<"goodbit: " <<
  cin.good();cout <<endl <<"eofbit : " <<
  cin.eof();
  cout <<endl <<"failbit: " <<
  cin.fail();cout <<endl <<"badbit : " <<
  cin.bad();
  getch();
```

```
}  
/* Output : Enter Value: 47 goodbit:  
1eofbit : 0failbit: 0badbit : 0  
Enter Value(enter characters to see  
effect on flags): ghj :  
goodbit: 0eofbit : 0failbit: 1badbit :  
0*/
```

---

### *13.3.4 IO Stream Library – Header Files*

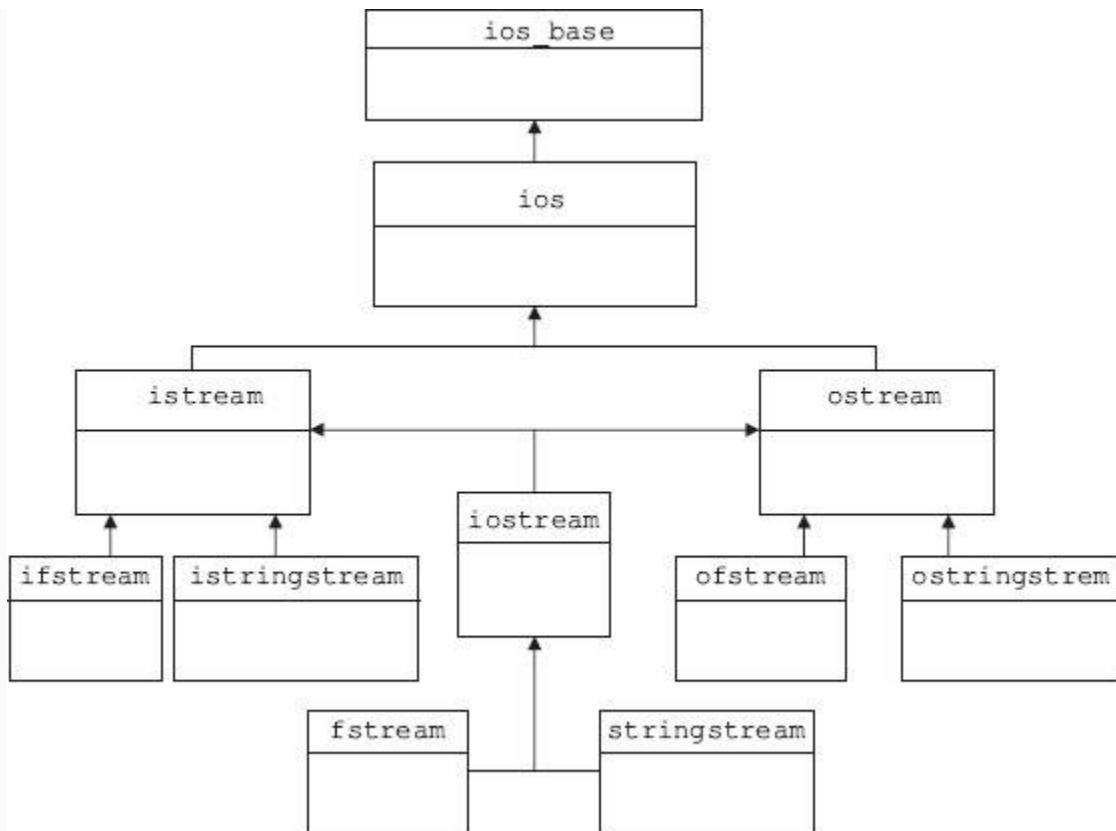
The `<iostream>` header file allows byte of data to flow from source to destination. Stream is independent of what data type is getting streamed or the source of destination.

C++ supports **`istream`** for input and **`ostream`** for output. IO stream library includes four types of predefined streams:

- `cin` : for standard buffered input
- `cout` : for standard buffered output
- `cerr` : for unbuffered error output. Works just like `cout`
- `clog` : for buffered log

`<iomanip>` header file consists of several functionalities to handle formatted output such as `setw()` `setprecision()`, etc.

`<fstream>` contains all the programs needed for file handling. The hierarchy of IO stream is shown in Figure 13.4.



**Figure 13.4** IO stream class hierarchy

## Example 13.4: `io1.cpp` Program to Show `Cin` and `Cout` at Work

```
#include<iostream>
using namespace std;
```

```

void main()
{double price; char title[30]; char
author[30];
    char * tell = " Enter the title of the
Book, Author and price :"; cout<<tell;
    cin>>title>>author>>price;
cout<<"\nTile of the book
:"<<title<<endl;
    cout<<"\nAuthor of the book
:"<<author<<endl;
    cout<<"\nPrice of the book
:"<<price<<endl;
}
/*Output :Enter the title of the Book,
Author and price :C++ ramesh 250.00
Tile of the book :C++ : Author of the
book :ramesh : Price of the book :250*/

```

---

## 13.4 IO Manipulators

Manipulators alter the status of streams. For using this feature we need to include the statement: `#include<iomanip>` in global section. For example, if we use a manipulator `setprecision(2)`, all the floating point variable output will be two digits after decimal point. A few of the important IO manipulator are:

---

```
cout<<dec<<intvar;decimal / hexa / octal
representation from integers
cout<<hex; cout <<oct;
cout<<setiosflags(ios::dec) Sets the
formatting bits as decimal
cout<<resetiosflags(ios::hex) Resets
already set flags as per format
specified
cout<<setbase(int n) :Sets bas to n
cout<<setw(int n) :specifies width for
output formatting
cout<<setfill("*"); :fills the unfilled
space set with setw() with fill
character
```

---

Character manipulators are shown in Table 13.2. Numeric and Data stream manipulators are shown in Tables 13.3 and 13.4, respectively.

**Table 13.2** Character manipulators

Manipulator	Affected Flag	Description
<code>setw(val)</code>	None	Sets the width of output field to the specified value
<code>setfill(char)</code>	None	Pads the unfilled width of the output with the specified character
<code>right</code>	<code>ios::right</code>	Right justifies the output
<code>left</code>	<code>ios::left</code>	Left justifies the output
<code>internal</code>	<code>ios::internal</code>	Left adjusts the sign and right adjusts the value

**Table 13.3** Numeric manipulators

Manipulator	Affected Flag	Description
dec	ios::dec	I/O uses decimal notation for input and output
hex	ios::hex	I/O uses hexadecimal notation for input and output
oct	ios::oct	I/O uses octal notation for input and output
scientific	ios::scientific	Display floating point number in scientific notation
fixed	ios::fixed	Display floating point number in fixed notation
setprecision(n)	None	Sets precision of floating point variables to n

**Table 13.4** Data stream manipulators



Manipulator	Affected Flag	Description
endl	None	Inserts newline character in the stream and flushes (sends) the buffer to output
flush	None	Flush the output buffer to the console
ws	None	Ignore white spaces in the input

---

```
cout<<endl; :new line to iostream and
flushes the buffer.
cout<<flush; :flushes ostream buffer
cout<<ends; :inserts null character at
the end
```

---

## Method 1: Using setiosflags to manipulate flags

---

```
cout<< setiosflags(ios::left);
cout << setw(10) << setfill('.') <<
"Hello" ;
```

---

## Method 2: Using Manipulators

---

```
cout << left <<setw(10)<<setfill('.')  
<<"Hello" ;
```

---

### Example 13.5: `io2.cpp` Program to Show `Cin` and `Cout` at Work

```
1.  #include<iostream>  
2.  #include<iomanip>  
3.  using namespace std;  
4.  void main()  
5.  {double price; char title[30]; char  
author[30]; int copies;  
6.  char * tell1 = " Enter the title of  
the Book :";  
7.  char *tell2 = " Enter the Author of  
the Book :";  
8.  char *tell3 = " Enter the price of  
the Book :";  
9.  char *tell4 = " Enter No of copies  
of the Book :";  
10. cout<<tell1<<flush;  
11. cin>>ws>>title; cout<<tell2<<flush;
```

```

cin>>author; cout<<endl3 <<flush;
12. cin>>price; cout<<endl4<<flush;
cin>>dec>>copies;
13. cout<<setiosflags(ios::left)
<<"\nTitle of the book :"<<title <<endl;
14. cout<<"\nAuthor of the book
:"<<author<<endl;
15. cout<<resetiosflags(ios::right)
<<"\nPrice of the book :"<<setw(10)
<<price<<endl;
16. cout<<"\n Number of copies
:"<<setw(10)<<copies<<endl;
17. }
/*Output: Enter the title of the Book
:C++ : Enter the Author of the Book
:Ramesh
Enter the price of the Book :250 :Enter
No of copies of the Book : 10000
Title of the book :C++ :Author of the
book :Ramesh : Price of the book :250
Number of copies :10000 */

```

**Li  
ne  
N  
o.  
10  
:**

shows that we need to use `flush()` to flush the buffer on to IO device.

<b>Line No. 11:</b>	instructs <code>ws</code> meaning accept white spaces while taking in title. Of course, once set, the flags continue to be set unless cancelled by altering the flag.
<b>Line No. 12:</b>	shows <code>flush()</code> and decimal setting. Line No 13 shows left justification.
<b>Line No. 15:</b>	shows <code>setw(10)</code> meaning width of 10. It also resets <code>iosflag</code> to right.

## 13.5 Flags

The arguments that can be passed to `setw` and `setiosflag` manipulators are shown below:

--	--

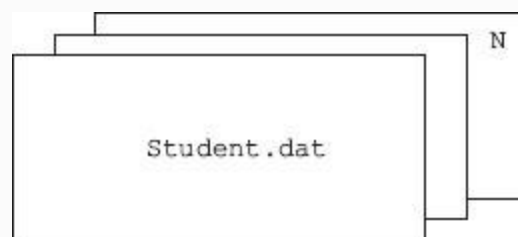
<code>ios::skipws</code>	skip white spaces in input stream
<code>ios::left</code> or <code>ios::right</code>	left or right justification
<code>ios::scientific</code>	
<code>ios::fixed</code>	follows decimal notation for floating point numbers
<code>ios::hex, ios::oct</code>	
<code>ios::showbase</code>	outputs the base number system
<code>ios::showpoint</code>	shows decimal point compulsorily.
<code>Ios::showpos</code>	shows + sign while displaying positive numbers

## 13.6 File I/O

We would come across files everywhere we go. For example, college holds a file for each of their students. Similarly municipality

holds files containing details of taxes to be paid by citizens. Indeed files are so common in our lives, C++ language and other languages support files. **What is a file?** A file is a collection of records. Figure 13.2 shows a file named `student.dat` with **n** records belonging to **n** number of students.

A record in a physical file is a data sheet wherein details of a student are recorded. There will be as many records as there are students. In a C++ file too, there will be records, again one for each student. Figure 13.5 shows a record.



**Figure 13.5** File and records

A record in turn contains fields. Figure 13.6 shows a record and fields contained therein.

Student Record 1	
Name:.....	rollNo:.....
marks[1]:..marks[1]:.....	marks[5]:...
Total.....	Grade:.....

```
char name[20]; // array of name with space for 20
characters
int rollNo;
float marks[5]; //marks as array for 5 subjects
float total;
char grade;
```

**Figure 13.6** Record and fields

### *13.6.1 File Types*

Files can be classified based on the way they are accessed from the memory as

**Sequential File:** All records are stored sequentially as they are entered. This type of file is best suited when we have to access all the records in sequence one after the other. Marks processing of a class is an example.

**Random Access File:** In this mode of access, a record is accessed using an index maintained for this purpose. It is like

browsing through a chapter and within the chapter a page of interest using the index provided at the beginning of the book.

**Direct Access File:** In this mode, the records are stored based on their relative position with respect to the first record. For example, record 50 will be 50 lengths away from the address of record 1. The main advantage of this mode of access is that there is no need to maintain indexes that would result in memory overhead. The disadvantage is that memory locations get blocked.

C++ language supports both sequential and direct access mode.

Files can be further classified as ***text files*** or ***binary files***. Normally, in a text file, data is stored using ASCII character code. Thus, to store 1234.5 in text mode, we would need 6 character spaces, i.e. 48 bits, whereas if you store it in binary mode, one would save a lot of memory as we will convert 1234.5 into binary and then store. Hence, for



storing intrinsic data of large numbers and sizes, binary mode is always preferred.

The classes required in connection with file handling in C++ are all kept in fstream header. Therefore, insert a statement :

```
#include<fstream>
```

To ***open*** a file for writing data on to it, we would create an object of ofstream like this:

---

```
ofstream outfile("Student.dat");
```

---

To ***open*** a file for reading data from it, we would create an object of ifstream like this:

---

```
ifstream infile("Student.dat");
```

---

We need to check if infile stream has been successfully allocated. In C++, logical NOT has been overloaded to check the stream file creation

---

```
if(!infile) {cerr<<"\n Sorry cannot open  
the file infile.."<<endl; exit(1);}
```

---

After use, every file that is opened needs to be closed like this `outfile.close()` ;  
`infile.close()` ;

### *13.6.2 Stream Operating Modes*

It is possible to control the stream files opening modes. We need to include these mode switches while creating the object through constructors.

<code>ios::app</code>	; appends data to the file at the end of file.
<code>ios::ate</code>	: opens the file and positions the reader or writer controller at the read head at the end
<code>ios::in</code>	: opens the file for reading
<code>ios::out</code>	: opens the file for writing
<code>ios::nocreate</code>	: fails to open if it does not exist already

<code>ios::noreplace</code>	: file should exist and also either of <code>ios::ate</code> or <code>ios::app</code> should be set.
<code>ios::trunc</code>	: truncates the file, if it already exists

We will write a program to copy a file's contents onto another file . In Example 13.6, we would use command line arguments feature. With commandline argument, you can execute the program from command line prompt **C:>\ iop io2.cpp io2copy.cpp** There are three arguments, namely, `argv[0]` = `io2.cpp` `argv[1]` = `io2.cpp` is the source. `argv[2]` = `io2copy.cpp` is the destination file. This feature allows us to execute the program without invoking IDE of Turbo C++ or VC++ directly from the command line prompt.

## Example 13.6: io3.cpp Program to Copy a File Through Command Line Arguments

```
#include<fstream>
#include<iostream>
using namespace std;
void main(int argc , char **argv)
{if ( argc<3)
    {cout<<"\n correct usage of
command line argument is : copyfile
inputfile
    outputfile.."<<endl; exit(0); }
    // open inputfile and connect it
to input stream inputstream
    ifstream inputstream(argv[1]);
    if(!inputstream)
    { cout<<"\n cannot open the input
file .."<<argv[1]<<endl; exit(1); }
    //open the output file and connect
it to outputstream
    ofstream outputstream(argv[2]);
    if(!outputstream)
    { cout<<"\n cannot open the output
file .."<<argv[2]<<endl; exit(1); }
    // now read from input file and
```

```
copy to output file
    char ch;
    while ( (inputstream.get(ch)) &&
outputstream ) outputstream.put(ch);
}
```

---

We have shown another version in which we obtain the file names from the user and execute copy source file onto the destination file in the solved example section, Ex 2 at the end of the chapter.

### **Example 13.7: io5.cpp Open a File in Append Mode, Append a Line**

---

```
#include<fstream>
#include<iostream>
using namespace std;
void main()
{   char filename[30];
    char text[80]; // for user input
    cout <<"\n Enter source file name...:
";
    cin>>filename;
```

```

        //Now lets open the file once again
for append mode
        ofstream outfile(filename,ios::app);
        if(!outfile)
        { cout<<"\n cannot open the input
file for appending.."<<filename <<endl;
        exit(1); }
        cout<<"\n Enter text for inputting
to file ..."<<endl; cin.ignore(1,'\n');
        cin.getline(text,80);
outfile<<text<<"\n";outfile.close();
        cout<<"\n completed writing to
output file"<<endl;
        // now lets read the file, we have
just outputted our text for confirmation
        cout<<filename<<endl;
        ifstream infile(filename);
        if(!infile)
        {cout<<"\n cannot open the input
file .."<<filename<<endl; exit(1);}
        char ch;
        while ( infile.get(ch)) // now read
from input file and copy to output file
        cout<<ch;
        cout<<"\n display process
completed.."<<endl;
        infile.close();
    }
/*Output:
Enter source file name...: io4.cpp :
Enter text for inputting to file ...
Hi we are trying out appending through

```

```
io5.cpp : completed writing to output
file
Hi we are trying out appending through
io5.cpp : display process completed..*/
Note that C++ supports all form of input
and output statements of C language. You
can freely use them in C++. We have used
the following statements in the program
    char text[30]; // buffer for user
input
    cin.getline(text,30); // getline
takes in to buffer text maximum of 30
characters
    // But new line character \n is
ignored and not taken in
    cin.ignore(1,'\n'); // ignores 1
character shown as second argument
```

---

## 13.7 Binary File

Normally, to facilitate direct reading on the console, the files are stored in text format. But for efficiency and saving the memory space, it would be far more efficient to store the data in binary form, i.e. in 0 s and 1 s. For example, a number 50595 is stored by text file as '5', '0', '5', '9', '5', whereas the binary form will allocate 2 or 4 bytes

depending on the hardware used and stores the number as binary equivalent >. In C++ a flag called `ios::binary` will specify the mode.

A binary file can be used to store all the intrinsic data types as well as user-defined data types like objects. We will use `write()` to write data on to binary files and `read()` to get the data from binary files.

---

```
Student std ( 20 , 70) ; // student
object with roll number and average
marks
fileobject.write( ( char*) & std ,
sizeof (std) );
fileobject.read( ( char*) & std , sizeof
(std) );
```

---

**Example 13.8: `io6.cpp` Program to Write a Class Called Student on to a File and Read Back the Contents of the File**



---

```
#include<iostream>
#include<fstream>
using namespace std;
//declare a class called Student
class Student
{ public: Student(int n, char *p, double
d) {rollNo=n,name=p, total =d;}
    ~Student(){}; // Default destructor
    // public access functions
    int GetRollNo() const { return
rollNo;}
    void SetRollNo ( int n) {
rollNo=n;}
    char * GetName() const { return
name;}
    void SetName( char *p) { name=p;}
    double GetTotal() const { return
total;}
    void SetTotal ( double d) {
total=d;}
private: int rollNo; char *name;double
total; // name is a pointer to name
};
void main()
{    char filename[80];
    cout<<"\n Enter file name...:";
    cin >>filename;
    ofstream
outfile(filename,ios::binary); // open
in binary mode
    if(!outfile) cout<<"\n Cannot open
```

```

iffile in binary
mode.."<<filename<<endl;
        exit(1); }
        Student std(5095,"Thunder",70.0);
// create a student object and write on
to file
        cout<<"\n Data of Student we have
created.."<<endl;
        cout<<"\n Student roll number :
"<<std.GetRollNo()<<endl;
        cout<<"\n Student name :
"<<std.GetName()<<endl;
        cout<<"\n Student total :
"<<std.GetTotal()<<endl;
        outfile.write( (char*) & std ,
sizeof std ); outfile.close();
        // now lets open the file and read
the content
        ifstream
infile(filename,ios::binary);
        if(!infile)
        { cout<<"\n Cannot open infile in
binary mode.."<<filename <<endl;
                exit(1); }
        // we will create a dummy std2 and
read the data from file on to std2
        Student std2(1,"No Name",0.0);
        infile.read( (char*) &std2 ,
sizeof(std2) );
        cout<<"\n Data of Student read from
file.."<<endl;
        cout<<"\n Student roll number :

```

```
"<<std2.GetRollNo()<<endl;
    cout<<"\n Student name :
"<<std2.GetName()<<endl;
    cout<<"\n Student total :
"<<std2.GetTotal()<<endl;
}
/*Output: Enter file name...:stdbinary:
Data of Student we have created..
Student roll number: 5095: Student name:
Thunder: Student total: 70
Data of Student read from file..
Student roll number: 5095: Student name:
Thunder: Student total: 70*/
```

---

## 13.8 Seekg() / Seekp() and Tellg() and Tellp() Functionality of C++

In C language, you have used, `fseek()` and `ftell()` to position the read & write cursor on the file. In C++, we will use `seekg()` and `seekp()` for get position and seek position. This is because in C++ the same stream is used for both input and output.

**Seekg()** alters the get position, i.e. the position from where we can read from a file.

**Seekp()** alters the position from where we can write.

---

```
        infile.seekg(256); // skip 256
bytes and position get cursor at
position 217
        outfile.seekp(1024); // position
the cursor for writing at byte position
1025
```

---

**tellg() and tellp()** functions on the other hand tell us about the position of read and write during our next read and write operation.

---

```
        int pos = infile.tellg() ; //
indicates the position of next read
operation.
        int pos = infile.tellp() ; //
indicates the position of next write
operation.
```

---

## **Example 13.9: io7.cpp Program to Write Student Object on to File and Use Seekg()/Tellg()**

---

```
#include<iostream>
#include<fstream>
#include<iomanip>
using namespace std;
const int max =3;
//declare a class called Student
class Student
{ public:
    Student(){}
    ~Student(){}; // Default destructor
    // public access functions
    void GetData(void);
    void DisplayData(void);
private: int rollNo; char name[20];
float mat,sci,eng,total,avg;};
void Student::GetData(void)
{ cout<<"\n Enter name : ";cin>>name;
cout<<"\n Enter id No : ";
cin>>rollNo;
    cout<<"\n Enter <Maths , Science and
English> Marks :"; cin>>mat>> sci>>eng;
    total = mat+sci+eng; avg=total/3.0; }
void Student::DisplayData(void)
{ cout<<"\n Name : "
<<setiosflags(ios::left)<<setw(20)
<<name;
    cout<<"\n Total : "<<setprecision(2)
<<setw(20)<<setiosflags(ios::fixed)
<<total<<endl;
    cout<<"\n Average : "<<setprecision(2)
<<setw(20)<<avg<<endl; }
```

```

void main()
{ Student std[2]; // two student objects
  char filename[30];
  int ch,pos,p,sz=0;
  cout<<"\n Enter filename..";
  cin>>filename;
  fstream inoutfile;

  inoutfile.open(filename,ios::in|ios::out
);
  do{cout<<"\n MENU ";
    cout<<"\n 1.Add a record\n 2.Display
nth record\n 3.Exit\n";
    cout<<" Enter choice of operation:
";cin>>ch;
    switch(ch)
    {case 1: if (sz>=max)
      { cout<<"\n out of bounds.."
<<endl; exit(1); }
      std[sz].GetData();

  inoutfile.write((char*)&std[sz],sizeof(s
td[sz]));
      sz++; // keep the count
      inoutfile.close();
      break;
    case 2: inoutfile.seekg(0); // go to
beginning
      cout<<"\nEnter the position of the
record to be displayed : ";
      cin>>p;
      if(p>max)

```

```

        {cout<<"\nPosition out of
bounds";break; }
        pos = (p-1) * sizeof(std) ;
        inoutfile.seekg(pos);
        inoutfile.read((char*)&std[p-
1],sizeof(std[p-1]));
        cout<<"\nDetails of record "
<<p<<endl;

        std[p-1].DisplayData();
        inoutfile.close();
        break;
    case 3: exit(1);
    }//end of switch
    }while( ch>=1 && ch<4);
} //end of main
/*Output: Enter filename...Student.dat :
MENU
1.Add a record 2.Display nth record
3.Exit
Enter choice of operation: 1 Enter name:
Ramesh Enter id No: 100
Enter <Maths, Science and English> Marks
:90 90 90
MENU Enter choice of operation: 2 enter
the position of the record to be
displayed: 2
details of record 2: Name: Gautam Total:
294.00 : Average: 98.00 */

```

---

## 13.9 Summary

1. IO streaming is flow of data from one device to another device. For example, from input device like console to memory or from memory to output device like console.
2. Input from keyboard and output to console are called standard input and output device.
3. There are operators like inserters and extractors (<< and >>) provided to work with IO stream objects like cin, cout, cerr to convert into ASCII to intrinsic data for storage and convert back to ASCII for outputting to standard IO devices.
4. The extractor represented by the ">>" symbol translates data sent by I/O devices in memory representation formats such as integers and floats.
5. iostream library object cin coupled with the extractor operator >> provides the user translation services and also the processing of special characters such as Carriage Return(CR), Space, Tabs and End of File(EOF).
6. IO streaming functions make use of buffers to cater to different speeds of IO devices and computer systems. Various flags are set by IO streaming function to inform to the user the status IO streaming.
7. High-level disk IO also called standard IO/stream IO: Buffer management is done by the compiler/operating system.
8. Low-level disk IO also called *system IO*: Buffer management is to be taken care by the programmer.
9. console IO: All the input and output functions control the way input has to be fed and the way output looks on standard output devices, i.e. screen.
10. Console IO is further classified as formatted IO and unformatted IO.
11. Unformatted I/O is also called low-level I/O. It offers the highest efficiency and throughput amongst all the other I/O mechanisms. Unlike formatted I/O, here input has no automatic whitespace, Tab, carriage return



- detection and processing, and also data is not formatted – the programmer has to interpret the data.
12. I/O library in C++ provides a mechanism to test whether a particular I/O operation has succeeded or failed. Each stream object maintains a set of flags which indicates the state of the stream after an I/O operation.
  13. C++ supports istream for input and ostream for output. IO stream library includes four types of predefined streams. cin, cout, cerr, clog
  14. IO manipulators alter the status of streams. For using this feature we need to include the statement:  
`#include<iomanip>` in global section.
  15. Flags are the arguments that can be passed to setiosflag and resetiosflags manipulators.
  16. File is a collection of records. Records are a collection of fields. Fields are the smallest units of data. Fields can be basic data types or user-defined or derived data.
  17. Sequential File All records are stored sequentially as they are entered.
  18. Random Access File: In this mode of access, a record is accessed using an index maintained for this purpose.
  19. Direct Access File: In this mode, the records are stored based on their relative position with respect to first record.
  20. Files can be further classified as **text files** or **binary files**. Normally, in a text file, data is stored using ASCII character code. In binary files, data is stored just like it is stored in memory. Hence, binary mode saves memory.
  21. The classes required in connection with file handling in C++ are all kept in fstream header. Therefore, insert a statement : `#include<fstream>`
  22. In C++, logical NOT has been overloaded to check the stream file creation. `if(!infile) {cerr<<"\n  
Sorry cannot open the file`

- `infile.. "<<endl; exit(1); } .` In C++, a flag called `ios::binary` will specify the mode.
23. Stream operating modes are used to control the stream files opening modes. Include these mode switches while creating the object through constructors.
  24. `Seekg()` alters the get position, i.e. the position from where we can read from a file.
  25. `Seekp()` alters the position from where we can write.
  26. `tellg()` and `tellp()` functions on the other hand tell us about the position of read and write during our next read and write operation.

## Exercise Questions

### Objective Questions

1. Which of the following statements are true in respect of unformatted IO?
  1. Input has automatic recognition of white space
  2. Low-level IO
  3. Input finishes with EOF
  4. Lowest efficiency
  1. i
  2. i and ii
  3. i, ii and iii
  4. d) ii, iii and iv
2. `<<` operator defined in IO Stream is called
  1. Inserter
  2. Extractor
  3. Translator
  4. Shift left
  1. i and iii
  2. i and ii
  3. i, ii and iii

4. ii, iii and iv

3. << operator defined in IO Stream is called

1. Insertor
2. Extractor
3. Translator
4. Shift left

1. i and iii
2. i and ii
3. ii and iii
4. ii, iii and iv

4. Which of the following statements are true with respect to high-level disk IO?

1. Also called standard IO
2. Buffer management by user
3. Also called stream IO
4. Operates on files

1. i
2. i, iii and iv
3. i, ii and iii
4. ii, iii and iv

5. Which of the following statements are true with respect to low-level disk IO?

1. Individual character processing
2. Buffer management by user
3. Input has automatic white space recognition
4. Operates with highest efficiency

1. i
2. i, iii and iv
3. i , ii and iii
4. i, ii and iv

6. Which of the following statements are true with respect to `read(char* str, int count)?`:

1. Reads count characters from Stream
2. Appends String Terminator character
3. It is part of unformatted IO
4. Returns error if EOF is encountered

1. i
2. i, iii and iv
3. i, ii and iii
4. ii, iii and iv

7. `setw ()` `setprecision()` functions are contained in

1. `<iostream>`
2. `<iomanip>`
3. `<cstring>`
4. `<fstream>`

8. Which of the following statements are true with respect to manipulators?

1. `endl` flushes the buffer on to console
2. Inserts a new line character
3. ignores white spaces in the buffer
4. i, ii and iii

1. i
2. i and ii
3. iv
4. ii and iii

9. Which of the following statements are true with respect to manipulators?

1. `ios::skipws`: skip white spaces in input stream
2. `ios::showpos`: shows position
3. `ios::showpoint` shows decimal point compulsorily
4. `ios::showbase` shows base

1. ii and iii
2. i, ii and iii
3. i and iv
4. i, iii and iv

10. `Seekg ()` can work relative to any position in the file TRUE/FALSE

11. For reading data from a random access file we use

1. `get()`
2. `getline()`
3. `readline()`

4. `read()`

12. `Write()` function cannot send to standard output stream `cout` TRUE/FALSE

#### Short-answer Questions

- 13. Define IO stream.
- 14. What is a byte stream?
- 15. Distinguish formatted and unformatted IO.
- 16. What do `peek()` & `ignore()` and `putback()` commands achieve?
- 17. What predefined IO streams are contained in IO stream library?
- 18. Define the terms file, record and field.
- 19. What are binary files? How do they save space in memory?
- 20. What is sequential access of a file?
- 21. Explain direct access method.

#### Long-answer Questions

- 22. Classify the IO stream functions.
- 23. Explain formatted and unformatted IO functions with examples.
- 24. Explain what is stream state and flags used by C++.
- 25. Explain IO stream class hierarchy.
- 26. Explain IO manipulators available in C++.
- 27. What flags can be sent as arguments to `setiosflag` and `resetiosflag` functions?
- 28. Explain file classification in terms of access of data.
- 29. Explain stream operating modes with suitable examples.
- 30. Explain `Seekg()` / `seekp()` and `tellg()` and `tellp()` functionality of C++.

#### Assignment Questions

- 31. Write a program to merge two files into a third file. Each file holds a specified number of strings.

32. Write a program to print telephone directory. Use a class called Telephone to hold the data and member functions for telephone directory to create the directory and the file.
33. Write a program to sort the file that holds a specified number of strings.

### **Solutions to Objective Questions**

1. d
2. a
3. c
4. b
5. d
6. b
7. b
8. b
9. d
10. True
11. d
12. False

# 14

## Generic Programming and Templates

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to understand*

- Using template classes employing template class member functions as well as non-template class member functions.
- Use type-specific friend functions and static member function and data.
- Use generic programs extensively for data structure implementation such as stack.
- Appreciate concepts of developing template libraries.

### 14.1 Introduction

Generic programming means writing programs that will work for any type of data, i.e. make the programs independent of the type of data. With generic programs, there will be no need to develop type-specific programs for each data type. Templates in C++ are implementation tools provided to write generic programs. For example, a data structure such as stack can be used for storing different types of data such as integers, floats, characters and strings of messages. Using template facility provided by the C++ language, a generic program can be written and stored in a directory and can be used by all programmers for different data types.

## 14.2 What is Generic Programming and Why Use Templates?

Template is a tool provided by C++ for achieving reusability of code. We can hence say that it is a productivity enhancement tool for the programmer. On many occasions, we come across problems in real life wherein the logic is the same but the



data set is different. Suppose that we need a function to sort the given data. The data could be data of integers or data of floats or data of double. Do we write separate functions for each of the data types or is there a tool or method wherein one function takes care of all the data types. Function template is the solution provided by C++.

When we consider the array data structure, do we need to write separate classes for each type of array objects such as array of Students, array of cars, or array of Employees? Class template allows us to write a single class template program and allow us to instantiate several arrays of different groups of objects. Therefore, we can conclude that C++ allows data types to be forwarded to function and class definition at run-time as parameters.

### 14.3 Template Classes

Suppose that we implement a stack – a last-in-first-out data structure – to store integers. Then we have modified through cut and paste strategy and implemented a stack

of floats. Now if we need stack of orders, what shall we do? Keep copying through cut paste! This is not reusability promised by C++. Class templates are very useful when the logic is the same and data sets are different. For example, if we have a class template for Stack class, we can store integers, floats, objects of students on to Stack. Similarly, an array expressed as template allows us to store an array of integers or an array of students, etc. The main advantage of template classes is that if you declare a class template, you can store it in a directory and use these definitions of class templates as and when required in program development, just by creating instances of objects as and when required. It is precisely this concept of reusability that is of immense value to developers. This aspect will be studied in more detail in the next chapter on standard template libraries.

Stack is the most widely used-user defined structure in problem solving. C++ templates allow us to implement a generic `Stack<T>` template that has a type parameter T. T can be replaced with actual types at run-time, for

example, `Stack<int>`, and C++ will generate the class `Stack<int>`. We can also create classes such as `Stack<float>`, `Stack<Student>` and `Stack<char>`. Define template once and use it to create classes of different intrinsic and user-defined data types. Incorporating changes is very easy since once the changes are implemented in the template `Stack<T>`, they are immediately reflected in all classes – `Stack<float>`, `Stack<Student>` and `Stack<char>`. Our next example amplifies these important concepts.

### **Example 14.1: Template Class – Stack**

A class template definition is just like a regular class definition. It is preceded by the keyword `template`. For example, here is the definition of a class template for `Stack`. Note that `T` is a parameter value which changes

with each instance of class we create. The word class is a keyword that specifies that T is a parameter that will be used to represent the parameter type (T). We can create instances of int, float and Students in the void main section as follows:

---

```
Stack<int> I; //Stack of integers
Stack<float> F; //Stack of floats
Stack<Student> stdStack; // Stack of
students
```

---

Note that we can use Stack<int> or Stack <float>, etc., as data types anywhere in the program just like you use normal data type, for example, to pass arguments, to receive return values, etc. Stack class shown below is like any other class except that parameter type T is used as data type:

---

```
1. template <class T>
2. class Stack
3. {public:Stack(); //default
constructor
4. Stack(const Stack&); //Copy
constructor
5. ~Stack(){ delete [] stack_array;}
```

```

//destructor
6. Stack& operator=(const
Stack&); //Assignment operator
7. int Size() const;
8. int empty() const; //returns 1 if
stack is empty
9. T& top(); //returns top of stack
10. void pop(); //pops top of stack
11. void push(const T&); //pushes
12. template<class T>
13. friend ostream &
operator<<(ostream & out1, Stack<T>&S);
14. private:
15. T *stack_array; // Stack of data
integers, double, students
16. int tos;
17. int stack_size; // Stack size
18. };

```

---

Normally, if you declare and define any function, for example, say the constructor **Stack** **within the class** itself, like the constructor shown below, then there will be no need to use parameter type T:

---

```

Stack<T> :: Stack() { stack_array =
new T[max]; stack_size = max; tos = -1;}

```

But if you define outside the class, you have to use qualifiers as shown below:

```
template <class T>
Stack<T> :: Stack() { stack_array =
new T[max]; stack_size = max; tos = -1;}
```

---

**We define assignment operator as shown below:**

---

```
template <class T>
Stack<T>& Stack<T> :: operator=(const
Stack& S)
{ stack_array = new T[S.stack_size];
  tos = S.tos;
  stack_size = S.stack_size;
  for(int i =0 ; i<size; i++)
    stack_array[i] = S.stack_array[i];}
template <class T> // Size( )
function returns the size of the
stack. int Stack<T> :: Size() const {
return tos+1;}
```

---

**Prior to retrieving the data from the top of the stack, we need to ensure that stack is not empty. This is ensured by function empty() shown below:**

---

```
template <class T>
int Stack<T> :: empty() const
{if(tos == -1) return 1;
else return 0;}
```

---

Push operation in Stack data structure is like adding an element on to the top of the stack. There is a need to increment top of stack (tos) after each push operation.

---

```
template <class T>
void Stack<T> :: push(const T& val)
{ if( tos<stack_size-1 )
{ tos++;stack_array[tos] = val; }
}
```

---

In pop operation, we would eject the value from the top of the stack. This we would achieve by decrementing the top of stack (tos)

---

```
template <class T>
void Stack<T> :: pop()
{if(tos >= 0)tos--; }
```

In top() operation, we would use the value returned by the function for displaying the top of the stack. We would also decrement the tos, after top() operation:

```
template <class T>
T& Stack<T> :: top()
{ return (stack_array[tos--]); }
```

---

Also observe that we have declared << operator as friend of Stack class. As we want << operator to be friend of all instances of Stack class, we have declared as a template function as shown below:

---

```
template<class T>
friend ostream & operator<<(ostream &
out1, Stack<T>&S)
{out1<<"\n \t"<<"content"<<endl;
for ( int j=0 ; j<max; j++)
out1<<S.top()<<" : ";
}
```

---

Student class is a normal class that is provided to create an instance of students to template class. We would store the roll numbers on to stack, to be extracted on last-in-first-out basis. We have included operator << to be a friend of Student class

---

```
friend ostream & operator<<(ostream
&src, Student &S)
{src<<S.Display(); // to display the
roll no of the student
return src;}
```

---



In the next example, we would declare a student class . We would use this class to push instances of students on to Stack. We would also create instances of float Stack and int Stack to demonstrate the utility and power of the template.

**Example 14.2:**  
**stacktemplate1b.cpp**  
**Implementation of Stack Data**  
**Structure Using Templates**

---

```
#include<iostream>
#include<iomanip>
using namespace std;
const int max=10; // declares maximum
of array size
class Student
{public:
    Student(int roll); // constructor
with roll no input
    Student():rollNo(0){} //default
constructor
    ~Student() {} // default
destructor
```

```

        int Display() const {return
rollNo;}
        friend ostream &
operator<<(ostream &src, Student &S)
        {src<<S.Display();return src; }
private: int rollNo;
};
Student::Student(int roll):rollNo(roll)
{}
template <class T>
class Stack
{ public: Stack(); //default constructor
Stack(const Stack&); //Copy constructor
        ~Stack(){ delete [] stack_array;}
//destructor
        Stack& operator=(const
Stack&); //Assignment operator
        int Size() const;
        int empty() const; //returns 1 if
stack is empty
        T& top(); //returns top of stack
        void pop(); //pops top of stack
        void push(const T&); //pushes
        template<class T>
        friend ostream &
operator<<(ostream & out1, Stack<T>&S);
private:
        T *stack_array; // Stack of data
integers,double,students
        int tos; //top of stack
        int stack_size; // Stack size
};

```

```

template<class T>
ostream & operator<<(ostream &
out1,Stack<T>&S)
{out1<<"\n \t"<<"content"<<endl;
    for ( int j=0 ; j<max; j++)
        out1<<S.top()<<" : "; }
template <class T> //default constructor
Stack<T> :: Stack()
{stack_array = new T[max]; stack_size =
max; tos = -1;}
//copy constructor
template <class T> //copy constructor
Stack<T> :: Stack(const Stack& S):
stack_array(new T[S.size]),
    tos(S.tos),stack_size(S.stack_size)
{for(int i =0 ; i<stack_size; i++)
stack_array[i] = S.stack_array[i]; }
template <class T>
Stack<T>& Stack<T> :: operator=(const
Stack& S)
{ stack_array = new T[S.stack_size];
  tos = S.tos;
  stack_size =S.stack_size;
  for(int i =0 ; i<size; i++)
    stack_array[i] = S.stack_array[i];
}
template <class T>
int Stack<T> :: Size() const { return
tos+1;}
template <class T>
int Stack<T> :: empty() const
{if(tos == -1) return 1;

```

```

else return 0; }
template <class T>
void Stack<T> :: push(const T& val)
{if( tos<stack_size-1 )
{ tos++;stack_array[tos] = val; }
}
template <class T>
void Stack<T> :: pop()
{if(tos >= 0) tos--; }
template <class T>
T& Stack<T> :: top()
{return (stack_array[tos--]); }
void main()
{Stack<int> I; //Stack of integers
  Stack<float> F; //Stack of floats
  Stack<Student> stdStack;
  for ( int i=0;i<max;i++)
  { I.push(i+10); F.push(i+20);
stdStack.push(i+10000); }
  cout<<"\n The contents of the int
stack"<<endl;
  cout<<" "<<I;cout<<endl;
  cout<<"\n The contents of the float
stack"<<endl;
  cout<<" "<<F; cout<<endl;
  cout<<"\n The RollNos of the Student
from stack"<<endl;
  cout<<" "<<stdStack; cout<<endl;
  cout<<"\n endof program"<<endl;
}
/*Output :The contents of the int stack
Content 19 : 18 : 17 : 16 : 15 : 14 :

```

```

13 : 12 : 11 : 10 :
The contents of the float stack
    Content 29 : 28 : 27 : 26 : 25 : 24 :
23 : 22 : 21 : 20 :
The RollNos of the Student from stack
    content10009 : 10008 : 10007 : 10006
: 10005 : 10004 : 10003 : 10002 : 10001
: 10000*/

```

---

## 14.4 Function Templates and Passing of Arguments to a Function

Normal function accepting normal instance of data, for example, a function called FindMax accepting array of integers.

You must pass an integer instance of the array:

```
int FindMax(Array<int> &);
```

For a function that accepts Student array:

```
void FindMax (Array<Student>&);
```

A generic template function declaration can handle all kinds of arrays:

```
template <class T> void FindMax( Array<T>& );
```

A generic template function can also handle parameterized as well particular

instances of a template: `template <class T> void FindMax(Array<T>& , Array<float>);`

## 14.5 Template Friends

A template class can have friends. These friends can be

- Non-template friend class or function
- Template friend class or function
- Type-specific template friend function

We would refer the reader to Chapter 8 for general functions update. We have also seen in the previous example template friend function as friend operator `<<` to stack template class.

### *14.5.1 Non-template Function*

We can declare a class or function as friend to template class. In the example that follows, we would include a friend function called `FindSum()` that will add elements of integer array. This program will use a non-template friend function `FindSum()` that will work only on integer arrays because it is

a non-template friend function. We would also create instances of float stack and int stack.

**Example 14.3:**  
**stacktemplate2b.cpp A Non-**  
**template Friend Function**  
**FindSum() That Will Work Only on**  
**Integer Arrays**

---

```
#include<iostream>
using namespace std;
const int max=10; // declares maximum
of array size
template <class T>
class Stack
{ public: Stack(); //default
constructor
    Stack(const Stack&); //Copy
constructor
    ~Stack(){ delete [] stack_array;}
//destructor
    Stack& operator=(const
Stack&); //Assignment operator
    int Size() const;
```

```

        int empty() const; //returns 1 if
stack is empty
        T& top(); //returns top of stack
        void pop(); //pops top of stack
        void push(const T&); //pushes
        template<class T>
        friend ostream & operator<<(ostream &
outl,Stack<T>&S);
        int FindSum( Stack<int> I ) // I is
array of integers
        private: T *stack_array; // Stack
of data integers, double, students
        int tos;
        int stack_size; // Stack size
    };
    template<class T>
    ostream & operator<<(ostream &
outl,Stack<T>&S)
    {outl<<"\n \t"<<"content"<<endl;
        for ( int j=0 ; j<max; j++)
            outl<<S.top()<<" : ";}
    //definition of friend FindSum().Gets
access to private data
    int FindSum( Stack<int> I ) // I is
array of integers
    { int sum=0;
        for ( int j=0 ; j<max; j++)
            sum+=I.top();
        return sum;
    }
    template <class T> //default
constructor

```



```

Stack<T> :: Stack()
{stack_array = new T[max]; _size =
max; = -1; }
template <class T> //copy constructor
Stack<T> :: Stack(const Stack& S):
stack_array(new T[S.stack_size]),

tos(S.tos),stack_size(S.stack_size)
{ int i;
    for(i =0 ; i<stack_size; i++)
        stack_array[i] =
S.stack_array[i];
}
template <class T>
Stack<T>& Stack<T> :: operator=(const
Stack& S)
{ stack_array = new T[S.stack_size];
tos = S.tos;
stack_size =S.stack_size;
for(int i =0 ; i<size; i++)
stack_array[i] = S.stack_array[i];
}
template <class T>
int Stack<T> :: Size() const { return
tos+1;}
template <class T>
int Stack<T> :: empty() const
{if(tos == -1) return 1;
    else return 0;
}
template <class T>
void Stack<T> :: push(const T& val)

```

```

        {if( tos<stack_size-1 )
            { tos++;stack_array[tos] = val; }
        }
template <class T>
void Stack<T> :: pop()
{if(tos >= 0) tos--; }
template <class T>
T& Stack<T> :: top()
{ return (stack_array[tos--]); }
void main()
{ Stack<int> I; //Stack of integers
  //populate int stack
  for ( int i=0;i<max;i++)
  {I.push(i+10);}
  cout<<"\n Use friend function int
FindSum(Stack<int>)"
      <<"\n to calculate the sum
of int Stack"<<endl;
      int sum = FindSum(I);
      cout<<"\n Sum of integer Stack:
" << sum<<endl;
      cout<<"\n endof program"<<endl;
  }
  /* Output Use friend function int
FindSum(Stack<int>)
to calculate the sum of int Stack :
Sum of integer Stack: 145 */

```

---

## ***14.5.2 Template Friend Class or Function***

In Example 14.2, we have used extraction operator << as a template friend function of class Stack. In this section, we will introduce another very widely used data structure, i.e. array. Template declaration for Array class is given below:

### **Example 14.4: Template Declaration for Array Class is Given Below**

```
template <class T>
class Array
{ public:
    Array(int array_size=max); //
constructor
    Array(const Array&); //Copy
constructor
    ~Array(){ delete [] array;}
//destructor
    Array& operator=(const
Array&); //Assignment operator
    T& operator[] (int i) {return
array[i];}
    const T& operator[] (int i) const
{return array[i];}
```

```

        int Size() const { return
array_size;}
        template<class T>
        friend ostream & operator<<(ostream
& outl,Array<T>&S) ;
        private:
        T *array; // array of data
integers,double,students
        int array_size; // array size
};

```

---

Assignment operator =. First we would check if what we have on the left-hand side (lhs) and what we are equating with at the right-hand side (rhs) are one and the same. If same, there is no need to carry out the assignment operation. Hence, we return `*this` to called function Why return `*this`? You must appreciate that = operation allows chaining, i.e. we can write statement such as `A=B=C=D=25`. Then compiler first assigns `D=25` and chains to `C` and thence to `B` & `A`. Hence, we need to return reference so that chaining is possible. Then before copying from the rhs, it is a good idea to delete what is on the lhs. Therefore, we use `delete *array`:

---

```
template <class T>
Array<T>& Array<T> :: operator=(const
Array& S)
{if( this==&S)
    return *this; // source &
destination are same no need for =
operation
    delete *array; // delete
destination prior to copy from rhs
    array_size=S.Size();
    array=new T[array_size];
    for int i=0;i<array_size;i++)
        array[i]=S[i];
    return *this; // this is for
chaining = operator allows chaining
}
```

---

`int Size() const { return array_size; }` is public accessory function provided to access private data `array_size`.

**We have also declared << as a friend function of Array class**

---

```
template<class T>
    friend ostream & operator<<(ostream
& out1,Array<T>&S)
    {out1<<"\n \t"<<"Index"
```

```
<<"\t\t"<<"content"<<endl;
        for ( int j=0 ;
j<S.array_size; j++)

out1<<"\n\t"<<j<<"\t\t"<<S[j];
    }
```

---

Also note that if S[j] pertains to Student class, then it will invoke friend operator << defined in Student class.

---

```
int Display() const {return rollNo;}
friend ostream & operator<<(ostream
&src, Student &S)
{src<<S.Display();return src; }
```

---

We have included overloading of operator [ ] in Array class as shown below:

---

```
T& operator[] (int i) {return
array[i];}
const T& operator[] (int i) const
{return array[i];}
```

---

Please note that array has been declared as pointer belonging to data type T. Given off set or cell number of the array, operator [

] returns a reference of type T or const T. Because of these overloading operators, we could use statements like

---

```
for ( int i=0;i<I.Size();i++)
{ I[i]=i+10;
  F[i]=i+20.0;
  pstd=new Student(i+10000);
  stdarray[i]=*pstd;
}
```

---

Finally, observe that overloaded friend operator << has allowed us to use statements like **cout<<I;** in the main program. Student class is a normal class that is provided to create an instance of students to template class. Ex 2 at the end of the chapter gives a complete solution.

### *14.5.3 Type-specific Friend Function in Class Templates*

In the next program, we will use a template friend function `FindSum()` that will work on all types of arrays because its a template friend function. We will also declare

operator << (extraction operator) as friend of Stack class.

Note that we have placed both these functions above public and private declarations in Stack class. Friend in any case gets access to private data. Hence, it can be above public and private declarations of the class. This is one of the ways to declare and define friend functions.

Note further that we have declared friend functions to be **type-specific**. This means that integer Stack will be a friend of `Stack<int>` class only. Similarly, float Stack will be a friend of `Stack<float>` class. Hence, we have omitted template <class T> in our friend function definitions.

In the example that follows, we will also show you how to pass the template object as an argument to a function FindSum.

### **Example 14.5: stacktemplate3b.cpp**

#### **Type-specific Friend Functions**



---

```

#include<iostream>
using namespace std;
const int max=10; // declares maximum
of array size
template <class T>
class Stack
{friend void FindSum( Stack<T> I ) //
I is stack of integers
    { T sum=T(0);
      for ( int j=0 ; j<max; j++)
          sum+=I.top();
      cout<< sum<<endl;
    }
    friend ostream & operator<<(ostream &
out1,Stack<T>&S)
    {out1<<"\n \t"<<"content"<<endl;
      for ( int j=0 ; j<S.Size(); j++)
          out1<<S.top()<<" : ";
      return out1;
    }
public: Stack(); //default
constructor
      Stack(const Stack&); //Copy
constructor
      ~Stack(){ delete [] stack_array;}
//destructor
      Stack& operator=(const
Stack&); //Assignment operator
      oint Size() const;
      int empty() const; //returns 1 if
stack is empty

```

```

        T& top(); //returns top of stack
        void pop(); //pops top of stack
        void push(const T&); //pushes
        void SetTos(){ tos=max-1;}
private:
        T *stack_array; // Stack of data
integers,double,students
        int tos;
        int stack_size; // Stack size
};
        template <class T> //default
constructor
        Stack<T> :: Stack()
        {stack_array = new T[max]; stack_size
= max; tos = -1;}
        template <class T> //copy constructor
        Stack<T> :: Stack(const Stack& S):
stack_array(new T[S.stack_size]),

tos(S.tos),stack_size(S.stack_size)
        { int i;
          for(i =0 ; i<stack_size; i++)
            stack_array[i] = S.stack_array[i];
        }
        template <class T>
        Stack<T>& Stack<T> :: operator=(const
Stack& S)
        { stack_array = new T[S.stack_size];
          tos = S.tos;
          stack_size =S.stack_size;
          for(int i =0 ; i<size; i++)
            stack_array[i] = S.stack_array[i];

```

```

    }
    template <class T>
    int Stack<T> :: Size() const
    {return tos+1;}
    template <class T>
    int Stack<T> :: empty() const
    {if(tos == -1) return 1;
     else return 0;
    }
    template <class T>
    void Stack<T> :: push(const T& val)
    {if( tos<stack_size-1 )
     { tos++;stack_array[tos] = val; }
    }
    template <class T>
    void Stack<T> :: pop()
    {if(tos >= 0) tos--; }
    template <class T>
    T& Stack<T> :: top()
    { return (stack_array[tos--]); }
    void main()
    {Stack<int> I1; //stack of integers
     Stack<float> F1; //stack of floats
     // fill the data in to all three
stacks
     for ( int i=0;i<max;i++)
     { I1.push(i+10);
       F1.push(i+20.0);
     }
     cout<<"\n output using friend <<
operator"<<endl;
     cout<<"\n The contents of the stack

```

```

of int type"<<endl; cout<<I1;
    cout <<"\n The contents of the stack
of float type"<<endl; cout<<F1;
    //tos() function decrements tos for
each display till tos=-1
    //To Go to top of stack Call SetTos()
function for reusing Stack
    F1.SetTos(); I1.SetTos();
    cout<<"\nUse friend function T
FindSum(Stack<T>)"
        <<"\nto calculate the sum of T
type stack"<<endl;
    cout<<"\n Sum of integer stack :
";FindSum(I1);
    cout<<"\n Sum of float stack : ";
FindSum(F1);
}
/*Output: out put using friend <<
operator
    The contents of the stack of int type
        Content 19 : 18 : 17 : 16 : 15 :
    The contents of the stack of float
type
        Content 29 : 28 : 27 : 26 : 25 :
    Use friend function T
FindSum(Stack<T>)
    to calculate the sum of T type stack
    Sum of integer stack : 145 Sum of
float stack : 245 */

```

---

## 14.6 Templates and Static Member Functions and Member Data

While handling static member data and member functions, all the rule that apply to static declarations with respect to ordinary classes also apply for template classes and member functions. Static declaration in template allows each instant of template to have its own static data. For example, if we declare a static member data to stack like `numberOfStacks` in our array class, then each instantiation of Student array and integer array will have its own `numberOfStacks` to keep track of how many student arrays or integer arrays have been created.

**Example 14.6: stacktemplate5b.cpp  
Program to Demonstrate Static  
Member**

---

```

        //functions and static data
#include<iostream>
using namespace std;
const int max=10; // declares maximum
of stack size
class Student
{ public: Student(int roll); //
constructor with roll no input
        Student():rollNo(0){}
//default constructor
        ~Student() {} ; // default
destructor
        private: int rollNo;
};
Student::Student(int
roll):rollNo(roll){}
template <class T>
class Stack
{friend void FindSum( Stack<T> I ) //
I is stack of integers
{   T sum=T(0);
        for ( int j=0 ; j<max; j++)
sum+=I.top();
        cout<< sum<<endl;
}
        friend ostream & operator<<(ostream &
out1,Stack<T>&S)
{out1<<"\n \t"<<"content"<<endl;
        for ( int j=0 ; j<S.Size(); j++)
out1<<S.top()<<" : ";
        return out1;
}

```

```

    }
    public: Stack(); //default
constructor
        Stack(const Stack&); //Copy
constructor
        ~Stack(){ delete [] stack_array;}
//destructor
        Stack& operator=(const
Stack&); //Assignment operator
        int Size() const;
        int empty() const; //returns 1 if
stack is empty
        T& top(); //returns top of stack
        void pop(); //pops top of stack
        void push(const T&); //pushes
        void SetTos(){ tos=max-1;} // To
go to top of stack
        static int GetNumberOfStacks() {
return numOfStacks;}
    private:
        T *stack_array; // Stack of data
integers,double,students
        int tos;
        int stack_size; // Stack size
        static int numOfStacks; // static
data to keep track of number of stacks
    };
    // template instantiation of static
member
    template < class T>
    int Stack<T>::numOfStacks=0;
    template <class T> //default

```

```

constructor
    Stack<T> :: Stack()
    {   stack_array = new T[max];
        stack_size = max;
        tos = -1;
        numOfStacks++;
    }
    template <class T> //copy constructor
    Stack<T> :: Stack(const Stack& S):
stack_array(new T[S.stack_size]),

tos(S.tos),stack_size(S.stack_size)
    { int i;
        for(i =0 ; i<stack_size; i++)
            stack_array[i] =
S.stack_array[i];
            numOfStacks++;
    }
    template <class T>
    Stack<T>& Stack<T> :: operator=(const
Stack& S)
    { stack_array = new T[S.stack_size];
        tos = S.tos;
        stack_size =S.stack_size;
        for(int i =0 ; i<size; i++)
            stack_array[i] = S.stack_array[i];
    }
    template <class T>
    int Stack<T> :: Size() const { return
tos+1;}
    template <class T>
    int Stack<T> :: empty() const

```



```

    {if(tos == -1) return 1;
      else return 0;
    }
    template <class T>
    void Stack<T> :: push(const T& val)
    {if( tos<stack_size-1 )
      { tos++;stack_array[tos] = val; }
    }
    template <class T>
    void Stack<T> :: pop()
    {if(tos >= 0) tos--; }
    template <class T>
    T& Stack<T> :: top()
    { return (stack_array[tos--]); }
    void main()
    { Stack<int> I1; //stack of integers
      Stack<float> F1; //stack of floats
      Stack<Student> stdStack; // stack
of Students
      // fill the data in to all three
stacks
      for ( int i=0;i<max;i++)
      {I1.push(i+10); F1.push(i+20.0);
stdStack.push(i+10000);
      }
      // declare & fill the data in
Second set of Stacks
      Stack<int> I2; //Stack of integers
      Stack<float> F2; //Stack of floats
      Stack<Student> stdStack2; // Stack
of students
      // fill the data in to all three

```

```

stacks for second set
    for ( i=0;i<I2.Size();i++)
        { I2.push(i+10); F2.push(i+20.0);
stdStack2.push(i+10000);
        }
    cout<<"\n Display static data
using full class specification.."<<endl;
    cout<<"\n integers Stacks"<<"\t"
<<Stack<int>::GetNumberOf Stacks()
<<endl;
    cout<<" float Stacks"<<"\t"<<"\t"
<<Stack<float>::GetNumberOf Stacks()
<<endl;
    cout<<" Student Stacks"<<"\t"
<<Stack<Student>::GetNumberOf Stacks()
<<endl;
    cout<<"\n Display static data
using object say I2 , F2 , stdStack2.."
<<endl;
    cout<<"\n integers Stacks"<<"\t"
<<I2.GetNumberOfStacks()<<endl;
    cout<<" float Stacks"<<"\t"<<"\t"
<<F2.GetNumberOfStacks() <<endl;
    cout<<" Student Stacks"<<"\t"
<<stdStack2.GetNumberOfStacks() <<endl;
    }
    /*Output Display static data using
full class specification.
    integers Stacks 2
    float Stacks 2
    Student Stacks 2
    Display static data using object say

```

```
I2 , F2 , stdStack2..  
    integers Stacks 2  
    float Stacks 2  
    Student Stacks 2*/
```

---

## 14.7 Template Exceptions

For dealing with exceptions while using in conjunction with templates, the programmer has the following options:

- Declare the exception class outside the template class.
- Declare the exception inside the template class.

Recall from Chapter 11 on exceptions that we have defined an exception class `stackfullexception {}` and `stackemptyexception {}` with an empty body within the class definition so that we could use it as a tool to catch the exception and enter the catch block. The catch block, then, informs the user and makes necessary rectification or exits the program. Note that this exception class has no body. Its sole purpose is to give entry to catch block. As an example, while programming Stack operations, we can include two exceptions,

namely, `stackfullexception` and `stackemptyexception`, within the template class `Stack` as shown below:

---

```
class Stack
{public: .....
    // Exception classes
    class stackfullerror {};
    class stackemptyerror {}
};
```

---

When an exception object is thrown, control passes to catch block wherein corrective mechanisms are in place. Note that we have to use `Stack<string>` as we are using templates and `string` specifies the parameter type.

---

```
    catch(Stack<string>::stackfullerror)
    { cout<<"\nstack full exception .
      Cannot be pushed on to stack:"<<endl;
      cout<<"\nExiting the catch block of
      stackfull exception .."<<endl;
    }
```

Ex 3 at the end of the chapter gives complete solution.

---

## 14.8 Summary

1. Generic programming means writing programs that will work for any type of data, i.e. make the programs independent of type of data.
2. Template is a tool provided by C++ for achieving reusability of code.
3. Class templates are very useful when the logic is the same and data sets are different.
4. A generic template function declaration can handle all kinds of arrays `template <class T>void FindMax( Array<T>& )`.
5. A generic template function can also handle parameterized as well particular instances of a template: `template <class T> void FindMax( Array<T>& , Array<float> )`.
6. A template class can have friends viz non-template friend class or function, template friend class or function, or type-specific template friend function.
7. We can declare a class or function as a friend to template class.
8. We can also declare a type-specific friend function in class templates.
9. Template classes can have static member functions and member data.
10. Template classes can have exception classes also.

## Exercise Questions

### Objective Questions

1. Generic programming means

1. Program independence
2. Data-type independence
3. Hardware independence
4. Language independence

## 2. Generic programming relies on the concept of

1. Data binding
2. Abstraction
3. Polymorphism
4. Encapsulation

## 3. Templates in C++ achieves

1. Inheritance
2. reusability
3. Run-time polymorphism
4. Data hiding

1. ii
2. i and ii
3. i, ii and iii
4. i and iv

## 4. Creating a specific type of template is called

1. Objectization
2. Instantiating
3. Parameterization
4. Serialization

## 5. Templates are useful when

1. Logic and data are the same
2. Logic is the same and data are different
3. Logic and data differ
4. In all above cases

## 6. In template declaration: `template < class T >`

`class & T imply`

1. T is a parameter used
2. T is an object of class
3. Class is identifier of parameter of type T
4. T is a parameterized identifier used throughout the program

7. If a function is declared and defined within the class there is no need to use parameter type  
T. TRUE/FALSE
8. If a function is declared and defined outside the class, there is no need to use parameter type  
T. TRUE/FALSE
9. Passing a stack object to a function FindMax that is accepting Student stack as parameter, the prototype declaration can be written as

```
1. void FindMax ( Stack <T> & );  
2. void FindMax ( Stack & );  
3. void FindMax ( Stack <Student> & );  
4. void FindMax ( Student <Stack> & );
```

10. Correct syntax for declaring a inline template function is

```
1. Template < class T> inline FindMax(Stack <T> &)  
2. Inline template < class T> FindMax(Stack <T> &)  
3. Any of a and b  
4. None of a and b
```

11. When class has static member functions and a private static member data, each instant of class will have its own static member. TRUE/FALSE
12. A generic template function cannot be made specialized template function by overriding the function with a specific type of argument template function. TRUE/FALSE

#### **Short-answer Questions**

13. Distinguish between template and macro.
14. Differentiate function arguments and template arguments.
15. What is parameterization? Explain with examples.
16. How many types of friend declarations are possible with templates? Explain with examples.
17. Distinguish specific and general friends in case of template functions.

18. How do you pass template class as argument to a template function and non-template function?
19. Explain static member function and static member data usage in a template class declaration.

#### **Long-answer Questions**

20. Explain the syntax for declaring a template function within a class definition and outside a class definition.
21. Explain assignment operator and extraction operator working with class templates with example data structure array.
22. How do templates handle friend functions? Explain with examples.
23. How do templates handle static member functions and data?
24. Discuss the syntax for handling exceptions inside a template.

#### **Assignment Questions**

25. Implement == operator for Stack class discussed in the chapter.
26. Implement exception classes `ArrayOutOfBounds()` and `IOError()` for Array class discussed in the chapter. Include exception within template class and also outside the template class.
27. Expand the definition of Array class to include functionality such as `Traverse()`, `Insert()` and `Delete()`, and `sort()` with exception handling mechanism. Catch block should reallocate the memory double the size of the original allotment; copy the contents into enlarged array and resume the program.
28. Write a template based cpp for Queue class. Include operators for `==`, `<<` and `++` to indicate servicing of next element in the Queue. Include functionality such as `Display()` and `Enqueue()` to insert in the end of the



- Queue, Dequeue () to service the first element by displaying the element.
29. Write a template based cpp for LinkedList class. Include operators for ==, << and ++ to indicate servicing of next element in the Linked List. Include functionality such as Display() and AddFront() to insert in the front of the LinkedList, FindElem() to find the element given the position and element value.

### **Solutions to Objective Questions**

1. b
2. c
3. c
4. b
5. b
6. d
7. True
8. False
9. c
10. a
11. True
12. False

# 15

## Object-Oriented Programming with Java

### LEARNING OBJECTIVES

*At the end of the chapter, you should be able to understand*

- Downloading and installing JDK required to run your programs.
- Setting environment variables.
- Understanding what processes are involved in running a Java program.
- Running a Java stand-alone program.
- Running a Java applet with Java swing components.
- Using Industry standard Eclipse Development Tool.

## 15.1 Introduction

Java has been specially designed for the Internet. There are many features of Java that are ideally suited for calling it as language for the network. C and C++ are proven structured and objective-oriented languages but they are not built for network and the Internet. Java is a pure objective-oriented language and we can program in Java without using objects.

In this chapter, we will introduce you to the World Wide Web and discuss aspects of Java that make the language best suited language for networking applications. The objective-oriented features of Java are discussed in brief and portability and platform-independent features are discussed in detail. Care has also been taken to provide examples of methods and programs to make you comfortable with the programming environment of Java.

## 15.2 Internet and the World Wide Web

Java has been specially created as a network language. There are two terms closely

connected with Java programming. They are Internet and the World Wide Web (www). They are NOT synonymous and are NOT the same.

Firstly, **network** is a collection of computers either homogenous or heterogeneous.

**Internet** is a collection or grouping together of networks. We can say that the Internet is a network of networks, comprising of millions of computers. These computers communicate with each other using Internet protocols.

**Protocol** is set of rules for communicating computers. One such popular protocol is HTTP (Hyper Text Transfer Protocol) which is widely used. FTP (File Transfer Protocol) is also another widely used protocol.

**World Wide Web (WWW)** is a way of accessing information on the Internet. It uses HTTP to exchange data amongst communicating computers. The services

provided by www uses Internet browsers such as Internet Explorer (IE), FireFox, and Google Chrome, etc. The information is exchanged using HTTP and web pages. Web page is a place where required information is stored. Web pages are linked through hyperlinks.

Who is the boss – Internet or www? It is Internet who is the boss. www is a method of Internet to exchange data over the Internet. Can you guess other ways to exchange data on the Internet. Yes, email is another popular method. It uses SMTP (Simple Mail Transfer Protocol).

### 15.3 C and C++ are Around – Then Why Java?

C is a structured and procedure-oriented and powerful language designed for systems programming such as operating systems, compilers, data base programs, etc. C was originally designed as a language for UNIX operating system. C++ was developed as an object-oriented programming language with

very well-developed reusability features such as inheritance, etc. However, both these languages are NOT designed to work with the Internet. Both C and C++ are machine dependent. This means that programs will work only on computers that have the same hardware and OS as that on which programs have been originally written. In other words, C and C++ programs lack wider portability.

On the other hand, Java has been specially designed to work with the Internet. The Internet has heterogeneous computers interconnected through www. Hence, Java is designed to be platform independent. This means that it is hardware and OS independent. There is a popular saying about Java language: Write here and execute it anywhere! Sun Micro Systems indeed used "Write Once, Run Anywhere (WORA) " as its catchword.

## 15.4 Java Story

James Gosling and Patric Naughton, inventors of Java language, were originally looking for designing a universal remote that

would work with any set top box. They wanted it to be a “virtual machine” that would truly be machine (set top box) independent. Later, they added advanced features and called it as “oak” . It was also known for some time as “Green” and ultimately settled as “Java” . The idea by Gosling was to build a virtual machine and a language that would work truly as a machine-independent language with features of already popular C++ language notation. The design goals envisaged by inventors of Java are:

- It should be simple, object oriented, and familiar.
- It should be robust and secure.
- It should be architecture neutral and portable.
- It should execute with high performance.
- It should be interpreted, threaded, and dynamic.

Sun Micro first released Java 1.0 in 1995. All major web browsers incorporated and supported Java 1.0 . With rich platform-independent and easy to program features, Java soon became very popular. J2SE 1.2 was released in 1998–1999 to cater to different platforms. J2EE catered to

Enterprise Edition and J2ME catered to mobile computing requirements. Since 2006, Java is known by three names based on the segments they provide service, namely Java SE (Standard Edition), JavaEE, and JavaME.

## 15.5 Java Features

### *15.5.1 Portability or Platform Independent*

Machine-dependent languages, such as C and C++, use compilers that produce object codes that are highly machine/hardware and OS dependent, whereas programs written in Java run without any modifications on any other hardware or operating system. To achieve this feature, Java uses "byte code" and a virtual machine.

Byte code is interpreted by virtual machine (VM) for the host hardware. Users need to install JRE (Java Run-Time Environment) along with JDK for running Java applications. JRE is also available in web browsers for executing Java applets. What is the cost involved in getting all



important machine independence and portability? It is speed. Interpretation is slow compared to the compilation process. To increase the speed, Just in time Compilers (JIT compilers) were introduced that convert byte code into machine code. JDK shipped by Sun Micro Systems is a superset of Java compiler, Javadoc, and a debugger.

### *15.5.2 Automatic Garbage Collection*

Unlike C++ where in program had to keep track of dispersal of objects after use, Java resorts to automatic memory release for objects no more needed. Java uses automatic garbage collector to manage memory. So what is automatic garbage collector? Garbage collector is a software program that runs in the background and picks up and clears memory space occupied by the dead objects. It follows a predetermined algorithm to decide which objects have no remaining reference, i.e. they are no longer referred and frees the memory automatically. This process will

happen when the program is idle or when there is insufficient free memory on the heap to be freed.

### *15.5.3 Object-oriented Features*

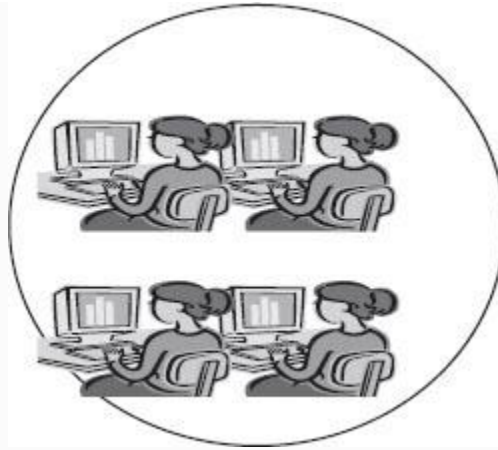
For detailed treatment on object-oriented features, we would strongly advise the reader to visit Chapters 1, 2, and 3. The succeeding sections give a brief overview of the concepts involved.

#### **15.5.3.1 Classes and Objects**

Object-oriented programming is a methodology in which we create cooperating objects that communicate with each other and accomplish the desired task. Java is a pure object-oriented language. You cannot write a Java program without a class. In fact program resides in a class. Java program uses objects and classes. So when we refer to object called Student we refer to both attributes(state) and behaviour. Attributes are also called *state* of an object.

**Class:** A collection of objects. We can also define as an array of instances objects. But

class can have member functions and member data. Here unlike array, a class can have different data types as its elements. Class defines abstract, i.e. hidden characteristics of an object including attributes and behaviour. A class called Student can be viewed as a factory producing instances of Object Student that have different attributes (i.e. individual data) and a common functionality (i.e. common functions) Attributes and functions provided by the class are called member data and member functions. In Java, member functions are called methods. A class allows us to encapsulate member functions and member data into a single entity called object. Figure 15.1 shows objects of students class.



**Figure 15.1** | Student objects

**Objects – Encapsulation:** In object-oriented programming like Java is a data primacy language, i.e. data is important and functions are not important. As per memory mapping, data is stored in data areas such as stack, free space, functions are stored in code area and there is a need to maintain strict control over the accessing of data by functions. Java achieves this control by using encapsulation feature. *Encapsulation is binding member data and calling function together with security classification, so that no unauthorized access to data takes place.* Java depends

heavily on access specifiers to maintain data integrity. The security access specifiers are ***public, private, protected, and default specifier package.***

**Public:** Member functions and data if any declared as **public** can be accessed outside the class member functions.

**Private:** Member data declared as **private** can only be accessed within the class member functions and data is hidden from outside.

**Protected:** Member data and member functions declared as **protected** are private to outsiders and public to descendants of the class in **inheritance** relationship. You will learn more about this in the chapter on Inheritance under C++ and the Java chapter that follows.

**Package (default):** You can access it from any other class in the same directory.

Encapsulation can now be defined as “Binding together the member functions and member data with access specifiers like private, public, and protected, package, etc. into object by the class.”

### 15.5.3.2 Method

In Java, class member functions are called Methods. A method is nothing but a code written to achieve a result. The syntax is:

```
ReturnType Function name (Argument List); //note the semicolon
    ↙       ↘       ↘
    int FindMax (int a, int b, int c);
void main () calls the function FindMax() by supplying the
arguments
and receives the result through return type.
    ans = FindMax ( a, b ) ; // a,b, and ans are all
    integer data types
```

**Arguments:** Number and type of parameters are called arguments. Parameters are also called formal parameters.

**Signature:** Name and arguments together are called the signature of the method. A method will have a return type which is outside the signature. The variables declared inside a method are called local variables and are never available anywhere outside the method. A method can be executed by sending a message to the object. Message comprises: a reference to the object, a dot ( . ), method name, and actual parameters (arguments). Values of actual parameters

are copied to formal parameters, a method gets executed and return value replaces the message. The local values are all lost.

#### 15.5.3.3 Polymorphism

Polymorphism means one method performing many different tasks depending on the users' requirements. This is one of the most powerful features of object orientation. There are two kinds of polymorphisms. Overloading and Overriding. Overloading means the same name but different signatures (number and type of parameters). When a message is sent, signatures are compared and best fit methods get loaded. This happens at run-time.

**Overriding** is a phenomenon which take place in inheritance. Overriding is commonly used to make methods more specific. Suppose both base class and class derived from base class have declared the same method with common signatures; then when a message is sent to an object, the method of derived class is loaded.

#### 15.5.3.4 Extending/Deriving New Classes

We can derive a new class based on the existing class. This is possible through the inheritance and containment property afforded by Java. Inheritance provides access to member data and member functions declared as protected to the descendant class.

#### **15.5.3.5 Inner Class**

##### **Class within a Class – Inner Class.**

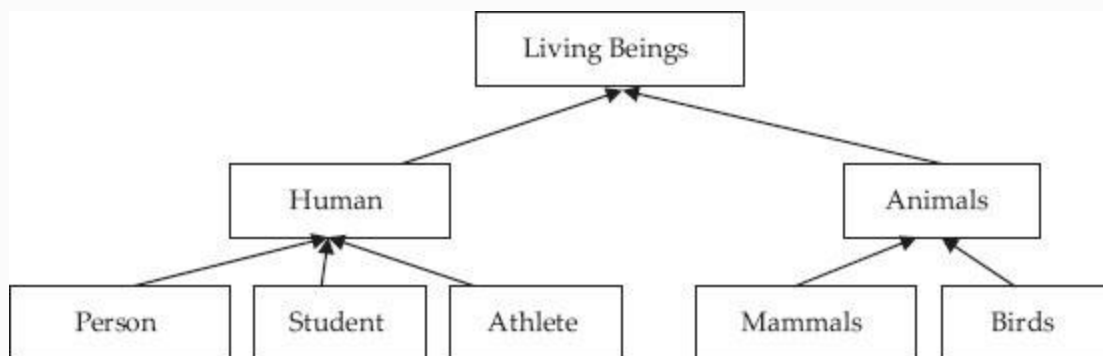
Inner class is one of the techniques provided by Java to achieve reusability of the code. Inner class means a class containing another class or more than one class. Inner classes cannot be declared as static.

#### **15.5.3.6 Inheritance and Class Hierarchy**

Reusability of code is one of the strong promises that are made by designers of Java and inheritance is the tool selected by Java to fulfill the promise of reusability. The concept of inheritance is not new to us. We inherit property, goodwill and name from our parents. Similarly, our descendants will derive these qualities from us. Study the inheritance class hierarchy shown in Figure



15.2. Human and Animal derive qualities from living beings. Professionals, students and Athletes are derived from Human. We say these three categories have inherited from Human. Class Human is called base class and Student and Athlete are called derived class. What can be inherited? Both member functions and member data can be inherited.



**Figure 15.2** Inheritance hierarchy

Have you noticed the direction of the arrow to indicate the inheritance relation? It is pointed upwards as per modelling language specifications.

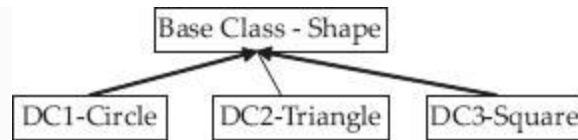
Inheritance specifies **is** type of relation. Observe that a Student is a Human. Similarly Mammal **is** Animal. Human is a **superclass** and Student is a **derived class**. Derivation from base class is the technique to implement **is** type of relation. A superclass can have more than one derived class.

When do we use Inheritance? You inherit so that you can derive all the functionality and member data from base class and derived class can add its own specialized or individualistic functionality. For example, Athlete derives all functionality and attributes of Human and in addition adds sports and Athletic functionality and attributes on its own.

Inheritance is a powerful tool in the hand of a programmer to define a new class from existing classes.

#### 15.5.3.7 Class as Abstract Data Type (ADT)

A class can also be called **Abstract Data Type (ADT)**. Refer to Figure 15.3.



**Figure 15.3** Hierarchical inheritance

We will declare a base class (ADT) called Shape. You will appreciate Shape has no definite shape, no definite area or perimeter or specific draw routine. Shape will hence declare methods with only names and no implementation details. In Java, these types of classes are called interfaces and methods are called ***abstract method***. An **abstract method** is a method that is *declared* but not *defined*. It is declared with the keyword ***abstract***, and has a header like any other method, but ends with a semicolon instead of a method body.

An **abstract class** is one that contains one or more abstract methods; it must itself be declared with the **abstract** keyword. A class may be declared abstract even if it does not contain any abstract methods. A non-

abstract class is sometimes called a **concrete class**. An abstract class cannot be instantiated because it has no solid methods. We will derive classes like Circle and Square and provide the **solid** implementation of the methods declared in the interface. Solid means that all methods will be implemented in each of the derived classes.

**Interface** is declared with the keyword `interface` instead of the usual keyword `class`. Interface can only contain abstract methods. However, using the keyword `abstract` for these methods is optional. **A class can extend to only one class but can implement several interfaces.**

#### 15.5.3.8 Generic Programming in Java

Generic programming is a facility provided by Java to achieve reusability of code. Generics can be applied to classes and methods. Class templates are very useful in achieving the reusability of code for different data types. No wonder then that Java relies heavily on Generic programming. Generic programming is very useful to abstract over types. For example, Java's Container types

present Java's Collection hierarchy, are very useful in implementing data structures often used in programming, such as Linked List, Queue, Stack, Searching and Sorting, etc. In a nutshell, Generic Programming improves productivity of the programmer.

#### *15.5.4 Easy to Learn and Excellent Documentation*

Since Java has done away with pointers, operator overloading, multiple inheritances etc., it is relatively easy to learn for a programmer. It follows C & C++ syntax which are already popular with programmers and widely used in the Industry. In addition, Java provides excellent inline documentation called `APIs` in `.html` format for providing instant help to the programmer. For this, Java provides three types of comment statements. Comments are included to enhance the readability of the program. Java allows both single-line comment statements and also multiline comment statements like C & C++

- **Single line comment.** These comment lines start with double slash : `//`  
Ex : `// HelloWorld.java , a Java program to print a line of text`
- **Multi Line Comment:** These comment lines are used when comments extend beyond a single line. They start with `/*` and end with `*/`  
Ex: `/* Inheritance.java :The program introduces concepts of Inheritance and other related reusability features of Java */`
- **Java API Documentation Comments:** These comments are used to describe features of Java program. Java documentation feature depends on .html file format and produce .html file we need to use these documentation comments. They start with `/**` and end with `*/`.  
Ex: `/** Class definition to show extend feature*/  
/** Method to compute student grades */`

These .html files are called APIs (Application Program Interfaces). To produce these APIs we need to use Javadoc compiler. In Section 15.5.4, Example 15.1, we have shown the procedure for producing API document for a Java source program.

### *15.5.5 Byte Code*

Byte code is an instruction set of one byte (8 bits). Byte code may often be either directly executed on a virtual machine (i.e. interpreter), or it may be further compiled into machine code for better performance. These are employed because they reduce the dependence on hardware and operating systems. Byte codes are compact numeric codes and references (normally numeric addresses) which encode the result of parsing and semantic analysis and hence offer much better performance than by direct reading of source code. A byte code program can be parsed directly executing instructions one at a time by byte code Interpreter. These are portable codes. After parsing and semantic analysis, part code that requires repeated execution, etc., are handed over to Just in time compiler (JIT Compiler) to translate byte code into machine code at run-time. Thus, JVM is not portable but byte code is fully portable. Because of its performance advantage, Java language implementations execute a

program in two phases, first compiling the source code into byte codes, and then passing the byte codes to the virtual machine. Therefore, there are virtual machines for Java.

### *15.5.6 Java Virtual Machine (JVM)*

JVM is specific to hardware and Operating System. Each JVM will have the following components:

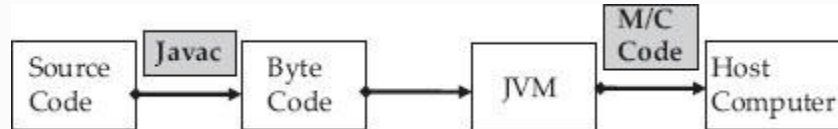
- Class Loader Mechanism for loading class files and interfaces.
- Execution Engine to execute instructions contained in class files.
- Run-time data areas to store class files, formal and actual parameters, return values, etc.
- Each instance of the Java virtual machine has one *method area* and one *heap*. These run-time data areas are shared by all threads running inside the virtual machine. All objects instantiated are stored on heap area.
- **Java Stack comprises several stack frames. Java allocates separate stack frames to each of the Java threads** (for now its just a new task).
- When a new thread comes into effect, it is allotted a Program Counter (PC) and its own Java Stack area. PC stores the address of the next instruction and Java Stack stores the state of Java Methods invoked. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value, and intermediate calculations. When a thread invokes a



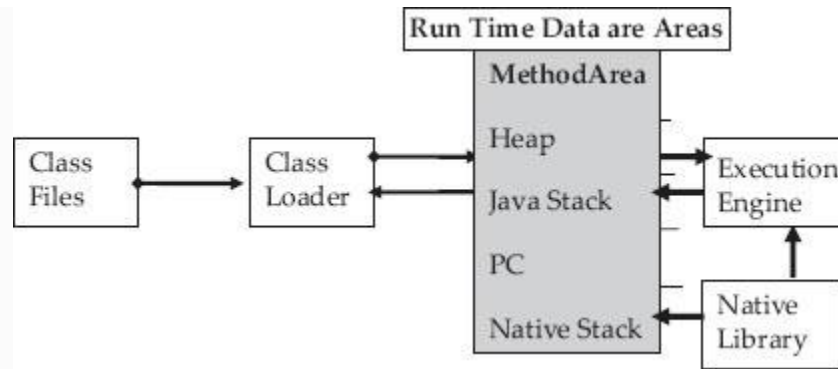
method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method.

- Native Stack area is machine dependent and keeps track of native method invocation.
- These areas are private to the owning thread. No thread can access the pc register or Java stack of another thread.

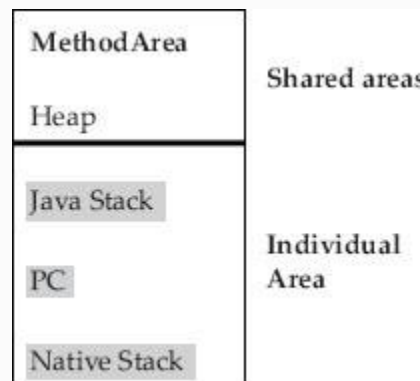
Figure 15.4 provides details of Java Development Environment. Figure 15.5 depicts the architecture explained above. Figure 15.6 gives details of run-time shared data areas.



**Figure 15.4** A Java Development Environment



**Figure 15.5** Architecture of JVM



**Figure 15.6** Shared and exclusive data areas. MethodArea and heap are shared by all Threads. Java Stack, PC are individual to each thread, which are shown in the shaded box

### 15.5.7 Comparison with C++

The main differences between C++ and Java are summarized in Table 15.1.

**Table 15.1** Differences between C++ and Java

<b>C++</b>	<b>Java</b>
------------	-------------

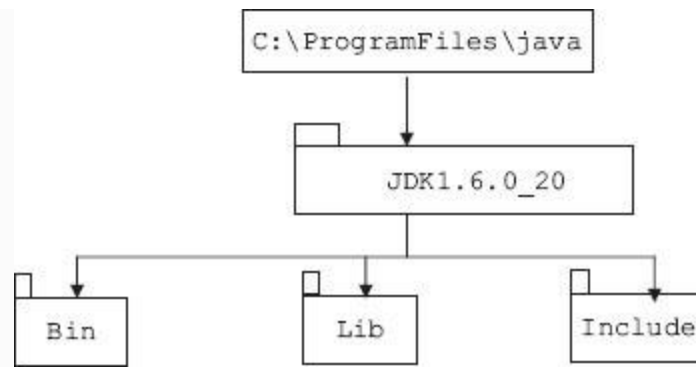
C++ is not pure objective-oriented language. It is possible to write C++ programs without using classes	Java is pure object-oriented programming language. Java program is a collection of classes
C++ supports pointers	Java does not support C/C++ style pointers
C++ primitive data types are objects	Primitive data types are not objects
In C++, the programmer has to free the heap memory occupied by the object when it is no longer required.	Freeing of heap memory of objects no longer referred is automatically carried out by Java's garbage collector
C++ supports operator overloading	Java does not support operator overloading
C++ supports multiple inheritance	Java does not support multiple inheritance
There are constructors and destructors	Java supports only constructor. Garbage collector does the job of freeing of objects no longer required
C++ supports two kinds of comments: a single line style marked with two	There are three different styles of comment: a single line style marked with two slashes (//), a multiple line style opened with a slash asterisk

slashes (//), a multiple line style opened with a slash asterisk (/*) and closed with an asterisk slash (*/)	(/*) and closed with an asterisk slash (*/) and the Javadoc commenting style opened with a slash and two asterisks (/**) and closed with an asterisk slash (*/)
Write once compile anywhere (WOCA)	Write once run anywhere / everywhere (WORA / WORE)
Allows direct calls to native system libraries	Call through the Java Native Interface and recently Java Native Access
Exposes low-level system facilities	Runs in a protected virtual machine environment
C++ supports native unsigned arithmetic	Java does not support native unsigned arithmetic
Pointers, References, and pass-by value are supported	Primitive data types always passed by value. Objects are passed by nullable reference
Supports class, struct, and union and can allocate them on heap or stack	Java supports only classes and allocates them to heap memory
No inline documentation is supported by C++	Javadoc is documentation supported by Java
C++ supports function pointers and function objects	No function pointers. Java uses interfaces and listeners to achieve the same purpose
C++ supports const keyword	Java supports final, a limited version for primitive data type

## 15.6 Developing First Java Application

### *15.6.1 Installing and Using Java Development Kit*

We will be using JDK1.6.0\_20 of Sun Micro Systems for developing and executing our programs. You can freely download the Java Development Kit (JDK1.6.0\_20) from `java.sun.com/javase/6/download.jsp`. Normally when you download JDK from Sun Micros web site, it gets downloaded to `C:\Program Files\java` unless otherwise chosen by you. After downloading, the JDK directory will look as shown in Figure 15.7. The **bin** directory contains both the compiler and the launcher. We will be demonstrating all our programs with JDK1.6.0 environment.



**Figure 15.7** Directory Structure after installing JDK 15.0\_20

### *15.6.2 Setting Path and Classpath*

It is very important to set environment for JDK 1.6.0\_20 so that compilation and execution take place very smoothly. There are two environment variables to be set. Of course, your programs will run without setting up these variables but every time we have to specify path and classpath which can be very tiring and cumbersome. Hence we recommend setting up these variables on a permanent basis, so that they are available at the time of booting the system.

Let us say that we want to use a folder called "Oopsjava" in C directory and

within the folder we have created a separate folder called "examples". Inside folder "examples", two subfolders, namely **bin** and **src**, are created. "**bin**" folder will house all our compiled code. In Java environment, file holding compiled code is called class file and will have .class as extension. Class file holds "byte code". "src" folder on the other hand will hold all our source programs. In addition, we would create a separate folder called "TestDriver" to hold all our test programs for class files we have written. These programs are also called **driver programs** or **interfaces**. Therefore, the path for storing all class files and source programs developed by us would be

---

```
C:\Oopsjava\examples\bin  
C:\Oopsjava\examples\src  
C:\Oopsjava\examples\TestDriver
```

---

Eclipse is an Industry popular IDE and can be used for developing Java applications. You can download from



[www.eclipse.org/downloads/](http://www.eclipse.org/downloads/). Usage of Eclipse is optional for you. Eclipse can be loaded into our folder “Oopsjava”.

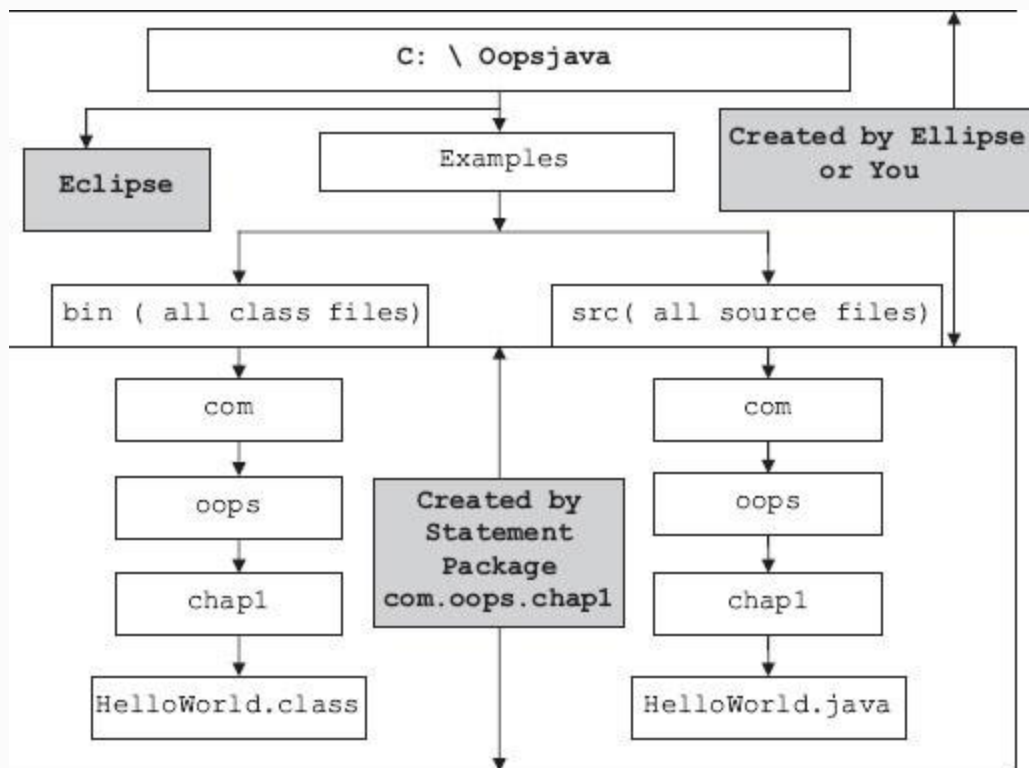
We also need to link up bin and lib directories of JDK1.6.0-20. Their paths would be

---

```
C:\Program
Files\Java\jdk1.6.0_20\bin // contains
all company supplied programs
C:\Program
Files\Java\jdk1.6.0_20\include; // win32
& header files
C:\Program
Files\Java\jdk1.6.0_20\lib // libraries
```

---

We can depict our directory structure pictorially as shown in [Figure 15.8](#). The upper part of the figure depicts the folders you have to create using the Operating system or created automatically when you use an IDE like eclipse. The lower part of the folders is automatically created by Java package statement. We will explain about Java package when we handle our first Java program.



**Figure 15.8** Organization structure

We need to set the path variable if we want to run Java executables such as `javac.exe`, `java.exe`, `javadoc.exe`, etc. from any directory without having to type the full path or classpath of the command. If you do not set these variables, you need to specify the full path to the executable every time you run it. By setting

path variable, we will have freedom to operate from any directory and calling any module or program without explicitly mentioning the path name.

Variable classpath, on the other hand, is a message from you to the Java environment, where to search for your class files.

Remember you have to set classpath only for the class file you have created. You need not bother about class files supplied by Java.

While setting these variables, note that dot “.” indicates current working directory.

## Example 15.1: How to Set Path and Classpath for Windows Platforms

---

Setting Environment Variables on Windows Platforms “

**Path:**

```
C:\Java\jdk1.6.0_20\bin;C:\Java\jdk1.6.0_20\lib;c:\java\jdk1.6.0_220\include;c:\Oopsjava\examples;c:\Oopsjava\examples\src;c:\Oopsjava\examples\bin;c:\ \Oopsjava\examples\TestDriver
```

Classpath :

```
.;c:\OopsJava\examples\bin;c:\OopsJava\examples\TestDriver
```

**Permanent Setting of variable:**

- Go to settings->control panel->system -> advanced-> set environment variable.

- Select path->edit : copy & paste path -> ok

- Select classpath ->edit : copy & paste ->ok

- Reboot the system

Alternately, you can set path and classpath each time you give the command

**Path:**

- C:\Program Files\Java\jdk1.6.0\bin  
> javac HelloWorld.java

**Classpath:**

- C:\OopsJava\examples>java -classpath c:\OopsJava\examples\bin> HelloWorld

To check whether CLASSPATH is set on Microsoft Windows XP, enter & execute

**C:> echo CLASSPATH**

- If CLASSPATH is not set you will get a %CLASSPATH% (MicrosoftXP).

---

## Example 15.2: How to Set Path and Classpath for Linux Platforms

Let us suppose that `jdk15.0_20` has been installed in directory `/usr/local/`

### **Linux:**

For C shell (csh), edit the startup file (`~/.cshrc`):

```
set path=(/usr/local/jdk1.6.0/bin
)
```

For bash, edit the startup file (`~/.bashrc`):

```
PATH=/usr/local/jdk1.6.0/bin:
export PATH
```

For ksh, the startup file is named by the environment variable, `ENV`.

```
To set the path:
PATH=/usr/local/jdk1.6.0/bin:
export PATH
```

For sh, edit the profile file (`~/.profile`):

```
PATH=/usr/local/jdk1.6.0/bin:
export PATH
```

Then load the startup file and verify that the path is set by repeating the Java command shown below:

---

```
For C shell (csh):  
% source ~/.cshrc  
% java -version  
For ksh, bash, or sh:  
% . /.profile  
% java -version
```

---

Alternately, you can set path each time you give the commands

### **Path:**

---

```
% /usr/local/jdk1.6.0/bin/>javac  
HelloWorld.java  
To find out if the path is properly  
set, execute:  
% java -version
```

---

This will print the version of the Java tool, if it can find it. If the version is old or you get the error **java: Command not found**, then the path is not properly set.

## Classpath in Linux:

We need to set our classpath to include three directories: current working directory denoted by a dot (.), `bin` directory and `TestDriver`.

The class path is the path that the Java run-time environment searches for classes. The class path can be set using either the `-classpath` option when calling Java tool such as `Javac` or `Javadoc` or `Java`, etc. by setting the `CLASSPATH` environment variable. The `-classpath` option is preferred because you can set it individually for each application without affecting other applications and without other applications modifying its value.

---

```
%javac -classpath
classpath1:classpath2:classpath
%javac -classpath
./usr/local/Oopsjava/examples/bin:

/usr/local/Oopsjava/examples/TestDriver:
```

---

Current working directory by default classpath search directory. Setting the CLASSPATH variable or using the -classpath command-line option overrides that default, hence we have included the current directory “.” in the search path.

---

You can also get the same result by executing the following command:

```
In csh % setenv CLASSPATH  
classpath1: classpath2...
```

```
In sh enter; CLASSPATH =  
path1: path2:... export CLASSPATH
```

To clear classpath already set from the previous commands

```
In csh: unsetenv CLASSPATH
```

```
In sh : unset CLASSPATH
```

To check if classpath is set correctly on Linux, execute the following:

```
% echo $CLASSPATH
```

If CLASSPATH is not set you will get a

```
CLASSPATH: Undefined variable  
error
```

### **Changing Startup Settings:**

If the class path is set at system startup then we should:

For shells csh , tcsh look for setenv command in .cshrc file



```
(~/ .cshrc): and enter
% setenv CLASSPATH
classpath1:classpath2...
For shells sh , ksh look into
.profile export command(~/ .
profile):
    CLASSPATH= classpath1:classpath2:
    export classpath
```

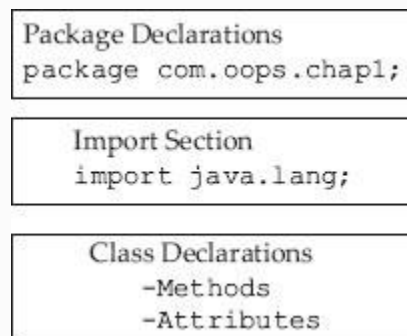
---

One final word about setting classpath. For .class files in an unnamed package, the class path ends with the directory that contains the .class files. For .class files in a named package, the class path ends with the directory that contains the "root" package (the first package in the full package name). Multiple path entries are separated by colons. We have explained package concept in Section 15.5.5.

So much for environment variables? Yes, understanding the environment variables setting is very important if you do not want to waste weeks in getting your advanced programs compiled and run correctly.

### *15.6.3 Java Program Structure*

Java is a pure object-oriented language. Remember that we cannot write anything outside the class. OOP is a creation of objects and achieves the desired result through communication and interaction amongst them. The sections in a Java program are shown in Figure 15.9.



**Figure 15.9** Program Structure

Therefore, all class files which are required are kept in a package. With a statement

---

```
package com.oops.chap1;
```

---

Refer to Figure 15.8. Setting classpath will take the control to

`c:\Oopsjava\exmples\bin`. Our package statement will create the directories `com\oops\chap1`.

Next important section is you need to import all the packages either supplied by Java or written and stored by you through the package statement explained above.

Finally, the structure will have a collection of classes. If you are engaged in writing a standalone application, then there will be class with `void main ()` statement. There can be more than one class. A Java program can also be an Applet-based program in which case the class will not have `void main()` but will have `init()` function.

#### *15.6.4 Java Documentation Comments*

The syntax is: `javadoc packagename`. Look at the organization chart in Figure 15.8. Our packages are kept in `src` with package `com.oops.chap1`. `javadoc` compiler and its usage in producing API documents are explained in Example 15.3.

## Example 15.3: How to Produce Java API Documents with Javadoc Compiler

We want to produce API documents to our package `com.oops/chap1`. These are situated in our `src` directory( refer to [Figure 15.8](#)). Go to `src` directory and type

```
C:\OopsJava\examples\src>javadoc  
com.oops.chap1
```

```
Loading source files for package  
com.oops.chap1...  
Constructing Javadoc information...  
Standard Doclet version 1.6.0_20  
Building tree for all the packages and  
classes...  
Generating com/oops/chap1/\ClassX.html...  
Generating com/oops/chap1/\ClassY.html...  
Generating  
com/oops/chap1/\HaiWorld.html...  
Generating
```

com/oops/chap1/\HelloWorld.html...  
Generating  
com/oops/chap1/\VrHello.html..  
Generating  
com/oops/chap1/\VrHelloIndia.html...  
Generating com/oops/chap1/\package-  
frame.html..  
Generating com/oops/chap1/\package-  
tree.html...  
Generating com/oops/chap1/\package-  
summary.html...  
Generating constant-values.html...  
Generating serialized-form.html...  
Building index for all the packages and  
classes... Generating overview-tree.html...  
Generating index-all.html... Generating  
deprecated-list.html...  
Building index for all classes...  
Generating allclasses-frame.html...  
Generating allclasses-noframe.html..  
Generating index.html...  
Generating help-doc.html.. Generating  
stylesheet.css...  
All the above documents with extension  
.html are produced by Java-doc compiler  
and stored at  
C:\oopsjava\examples\src>.Go ahead and  
get acquainted with Javadoc API. As a  
professional programmer you need to know  
the organization and method of  
presentation of Java documentation.

---

### *15.6.5 Java Development Environment*

Before we execute our program there is a need to understand the Java development environment (refer to [Figure 15.4](#)).

**Step 1: Edit:** Edit the source program using any of the editors notepads. For example, our program will have a main class "HelloWorld". Hence our Java program should also be named HelloWorld.java. Remember that we want to use package com.oops.chap1 as explained in [Figure 15.8](#). Hence move to directory C:\Oopsjava\examples\src\com\oops\chap1 and create the required source file.

```
cd
C:\Oopsjava\examples\src\com\oops\chap1> edit HelloWorld.java
```

**Step 2: Compile:** We need to store all our compiled code in

`C:\Oopsjava\examples\bin\com\oops\chap1`. Therefore from the directory

`C:\Oopsjava\examples\src\com\oops\chap1`. Compile the source file and redirect the class file into

`C:\Oopsjava\examples\bin\com\oops\chap1`. For redirecting you have to use `-d` option and provide the complete path to bin directory as shown below:

```
C:\OopsJava\examples\src\com\oops\chap1>javac -d
```

```
c: \Oopsjava\ examples\ bin  
HelloWorld.java
```

Compiler creates byte codes and stores then in a file with name `HelloWorld.class` at

```
c:\Oopsjava\examples\bin\com\oops\chap1.
```

**Step 3: Loader:** Class Loader ;Loads the byte codes (Class file) into primary memory

**Step 4: Byte code Verifier:** It checks if the byte codes are all ok and do not override Java's security restrictions.

**Step 5: JVM:** (Java Virtual Machine) calls interpreter and Just in time (JIT) compiler and converts the byte codes into object codes that local machines understand. JVM is a combination of interpreter and compiler. JVM as it interprets looks for "hotspots", i.e. parts of the program that are repeated, for example, a loop executing several times. In such cases, the JIT compiler converts them into machine code and for others, interpreter of JVM simply interprets the byte code.

### *15.6.6 Our First Java Application*



As our first Java application, we want to simply display a message: "HelloWorld, Welcome to Object-oriented Programming in Java".

### **Example 15.4: HelloWorld.java: Our First Java Application**

```
1. // A program HelloWorld.java to
   print a line
2. package com.oops.chap1;
3. public class HelloWorld
4. {
5.     public static void main(String[]
args)
6.     {
7.         System.out.println("Welcome to
Object Oriented Programming in Java ");
8.     }
9. }
```

---

## **Welcome to Object-oriented Programming in C++ and Java**

**L** package: As a developer, you would like to keep  
**i** all your work together, so that you can reuse  
**n** them in future if needed. We would like to keep  
**e** all our work as a "package". Suppose you are  
**N** working in a company called "oops.com",  
**O** then it is customary to create package as  
**.** "com.oops". Now we are writing programs for  
**2** Chapter 1. Hence we will name our package as  
**:** "com.oops.chap1". More details about  
package later in Chapter 19 (refer to Figure  
15.8). Path variable will link up to  
c:\Oopajava\example\src. It is your  
package statement which will link to  
com.oops.chap1.

**L** Source file must be named after the public  
**i** classes they contain. Hence we have named our  
**n** Java source program as HelloWorld.java.  
**e** The source file can contain only one public class  
**N** and several non-public classes.

**3** Keyword public means that methods inside a  
**:** public class can be called outside the class  
hierarchy. The hierarchy of the class is the name  
of the directory in which .java file resides.

Keyword static means that the class can be  
invoked directly without using an instance of

the class to invoke the method "main". This facility is required because normally you can invoke a method only through an instance of the object of the class. If we can invoke method main without instance of object, we can instantiate objects inside `main()` and achieve the desired result. Static just does this job. Note that `main()` is not a keyword of Java. Just that `main()` is called by Java to launch the program and pass control.

Void means that the main method does not return any value. Note also that void is a data type.

**String args:** The main method accepts an array of Strings objects of variable length as arguments normally through command line. Number of arguments, String objects, is specified by args.

**L  
i  
n  
e  
N  
o  
.  
7  
:** *System* class has defined a public static called *out*, which is an instance of `PrintStream` class and provides several methods for printing out data to `Standard.out`, such as `println()`, etc. The string "Welcome to Object-oriented Programming in Java" is automatically converted to `String` object by the compiler.

---

To produce the above output, Type/Edit the source program.

```
C:\OopsJava\examples\src\com\oops\chap1\ edit HelloWorld.java
```

Compile `WelcomeWorld.java` using the command

```
C:\OopsJava\examples\src\com\oops\chap1>javac -d
```

```
c:\Oopsjava\examples\bin  
HelloWorld.java
```

Now go to bin directory and type the command to execute Java program giving

the full package path.

```
C:\OopsJava\examples\bin>java  
com/oops/chap1/HelloWorld
```

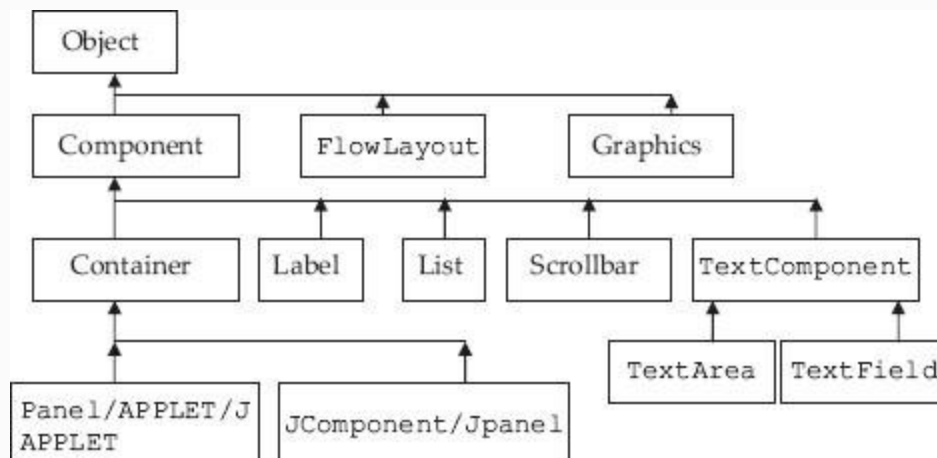
```
Welcome to Object Oriented  
Programming in Java
```

Note that since it is a simple program, we have executed it from bin directory. For large programs or programs with several classes, we would write a driver program and place the class program in directory `TestDriver`. Then we need to execute Java command from `TestDriver` directory.

### *15.6.7 Application with Swing Components*

Java extended package called Javax houses several graphical libraries which would greatly enhance the graphics components. One such facility is a dialog box for obtaining the input from the use and

displaying the message. Broad class hierarchy of JComponents is shown in Figure 15.10 for basic understanding. We have shown only the components that we are going to use in the immediate sections. More details can be found in Chapters 22 and 23 on Graphics and Swing Components. Example 15.5 deals Java application that uses dialog boxes of swing components in Java.



**Figure 15.10** Class hierarchy of graphics in Java

## **Example 15.5:**

### **HelloWorldDialog.java: Java Application with Dialog Boxes Using Swing**

```
//HelloWorldDialog.java
//We will use Dialog box offered
in swing components
1. package com.oops.chap1;
2. import javax.swing.JOptionPane;
3. public class HelloWorldDialog {
4.     public static void main(String[]
args) {
5.         String input =
JOptionPane.showInputDialog("Enter your
name?");
6.         // create the message
7.         String message =
8.         String.format("Welcome %s to
Object oriented programming in
9. Java",input);
10.        // display the message
11.
JOptionPane.showMessageDialog(null,
message);
```

```
}  
}
```

Class `JOptionPane` in swing package is a predefined class which provides dialog boxes to be readily inserted into your program.

`JOptionPane.showInputDialog("Enter your name?");` produces a dialog box for user input.

<b>L i n e N o . 7 &amp; 8 :</b>	<code>String message = String.format("Welcome %s to Object oriented programming in Java", input);</code> embeds input name in to message of String class. Finally Line No. 10 displays output again through Dialog box.
--	---

To obtain output Type/Edit the source program.



**Table 15.2** Escape characters and their functions

Escape Characters	Function	Escape Characters	Function
<code>\n</code>	new line	<code>\t</code>	tab
<code>\r</code>	Carriage return	<code>\\</code> backlash	To print \ character
<code>\"</code>	Double quote	<code>System.out.println(\"\\\"Ramesh\\\"");</code>	Output "Ramesh"

```
C:\Oopsjava\examples\src\com\oops\chap1\ edit
```

```
HelloWorldDialog.java
```

Compile `WelcomeWorld.java` using the command.

```
C:\OopsJava\examples\src\com\oops\chap1>javac -d
```

```
c:\Oopsjava\examples\bin  
HelloWorldDialog.java
```

Now go to bin directory and type the command to execute Java program giving the full package path.

```
C:\OopsJava\examples\bin>java  
  
com/oops/chap1/HelloWorldDialog  
a
```

## Dialog Boxes



**Caution/Note:** We are providing detailed compile and run instructions only in Chapter 1. We expect you to practice these commands well. In the succeeding chapters, we provide example programs only. Now that you have practiced two programs and learnt the edit, compile and load and run in command screen, we will introduce you to Java programming using "eclipse" JDE.

### *15.6.8 Eclipse-integrated Development Environment*

We have loaded `eclipse.exe` in  
`C:\OopsJava\eclipse-jee-galileo-SR2-win32\eclipse` and execute  
`eclipse.exe`

**Step 1:** File → new →javaproject  
..> project name : examples2  
//enter project name

---

Execution environment :java SE 1.6 //  
select from drop list  
Create separate folders for source &

class files // select the check box  
Finish.

At this stage you can see that required  
folders : examples2 and bin , and  
src have been created by eclipse at  
C:\OopsJava\examples2

---

## Step 2: File → new → class

---

Observe that you have chosen  
src folder: examples2/src package :  
com.oops.chap1  
class name as Calculator with  
java.lang.Object as its Super  
Application with public static void  
main()  
Inherit abstract methods  
Click : finish

---

“eclipse” will provide basic program  
structure. Type in the program shown in  
Example 15.6. Observe that eclipse would  
have created your package by creating  
additional folders in bin and src of your  
project directory examples2. The package  
you have asked is “com.oops.chap1”.  
Hence your Calculator.java program

will be positioned at

c:\Oopsjava\examples2\src\com\oops\chap1 directory. Similarly, class

file will be positioned at

c:\Oopsjava\examples2\bin\com\oops\chap1 directory. Run as Java

Application or Java applet as the case may be. You have two options: (1) Go through run command or (2) click run on the tool bar.

## Example 15.6: Calculator.java: Java Applet With Swing

```
1. package com.oops.chap1;
2. import javax.swing.*; // for
   JOptionPane
3. public class Calculator
4. {
5.     public static void main(String[]
   args)
6. {
7.     String number1,number2,choice; // two
   numbers from the user from console
```

```
8. int num1,num2,result,ch; // we need
these data to do internal calculations
```

---



```
9. //input data as strings
10.
number1=JOptionPane.showInputDialog("Ent
er first number");
11.
number2=JOptionPane.showInputDialog("Ent
er second number");
```

```
12. // get the choice of operation to be
    performed**
13. choice =
    JOptionPane.showInputDialog(
14. "Enter 1 - Addition :\n 2 -
    Subtraction ");
15. //for doing calculations we need to
    convert to intrinsic data type say int
16. num1 = Integer.parseInt(number1);
17. num2 = Integer.parseInt(number2);
18. ch = Integer.parseInt(choice);
19. if (ch==1)
20. result=num1+num2;
21. else
22. //subtract
23. result=num1-num2;
24. //display result
25.
    JOptionPane.showMessageDialog(null,"The
    result is "+result,
    "Answer",JOptionPane.PLAIN_MESSAGE);
26. // return i.e. exit programme
27. System.exit(0);
28. }
29. } // end Calculator prog
```

---

## Output



<b>Line No. 7:</b>	Number1, number2, and choice have been defined as variables of string class of Java. All inputs we will get as String class in Java.
<b>Line No. 15, 17, &amp; 18:</b>	We will convert String variables into int variable of Java to facilitate internal computations. <code>num1 = Integer.parseInt(number1);</code> will convert String class variable to Integer class variable.
<b>Line No. 19 – 23:</b>	If statement will confirm if operation is addition or subtraction.
<b>Line</b>	Displays the result through message dialog box of <code>JOptionPane</code> . Null in the arguments



<b>No . 25:</b>	states that dialog box to appear in the center of the screen, Message type is plain. The other options of type of messages are: JOptionPane.ERROR_MESSAGE JOptionPane.INFORMATION_MESSAGE JOptionPane.WARNING_MESSAGE JOptionPane.QUESTION_MESSAGE
<b>Line No . 27:</b>	Is <code>System.exit(0)</code> returns control to Java Run-Time (JRE) . Argument 0 indicates exit without error. Return 1 implies unusual program termination and return.

### 15.6.9 Command Line Arguments

In this example, we will revisit our calculator program and execute it through command line arguments. Please observe the `void main()` method

---

```

        public static void
main(String[] args) {

```

---

Strings [] args means we can pass any number of arguments to `main()`. These are array of Strings . Accordingly, `args[0]` `args[1]` `args[2]` .... etc. each argument is separated by a space. There are three arguments: `number1` , `number2`, `operation` (Add or Subtract)

### **Example 15.7: Calculator2.java: A Calculator Program Command Line Argument**

```
1. package com.oops.chap1;
2. import javax.swing.*;
3. public class Calculator2 {
4.     public static void main(String[]
args) {
5.         int num1 = Integer.parseInt(args[0]);
6.         int num2 = Integer.parseInt(args[1]);
7.         int ch = Integer.parseInt(args[2]);
8.         int result;
9.         if (ch==1)
10.            result=num1+num2;
11.         else
```

```
12. //subtract
13. result=num1-num2;
14. //display result
15.
OptionPane.showMessageDialog(null,"The
result is "+result,
"Answer",JOptionPane.PLAIN_MESSAGE);
16. // return i.e exit programme
17. System.exit(0);
18. }
19. } // end Calculator prog
```

---



Compile Calculator2.java using the command.

```
C:\OopsJava\examples\src\com\oops\chap1>javac -d
```

```
c:\Oopsjava\examples\bin
Calculator2.java
```

Now go to bin directory and type the command to execute Java program giving full package path with arguments: 10 10 1

```
C:\OopsJava\examples\bin>java  
com/oops/chap1/Calculator2 10  
10
```

A common mistake everybody does at the stage is to forget to give complete package path: java com/oops/chap1/Calculator2. If you forget this, you will encounter "class not found error".

## 15.7 Summary

1. Java is a networking language. The Internet is a collection of cooperating networks. A network is a collection of cooperating computers.
2. WWW (World Wide Web or simply web) is a way of accessing information from the network. We use HTTP/FTP for retrieval of information from the web. Email (through Simple Mail Protocol, SMP) is another important and widely used method to exchange information on the web.
3. Java is platform independent, i.e. it is independent of underlying hardware and operating systems that are controlling the hardware resources.
4. Java was originally named as “OAK” by its inventors James Gosling and Patrik Naughton.
5. Extending/Deriving new classes from existing classes, containment through inner classes, Extending through Inheritance mechanism.
6. Single, multi level , and hierarchical inheritance are allowed by Java . Java does not support multiple inheritance.
7. A class with abstract method is called abstract class with keyword abstract. An abstract class need not necessarily have abstract methods.
8. **Interface** is declared with keyword interface instead of usual keyword class. Interface can only contain abstract methods. However, using keyword abstract for these methods is optional.
9. **A class can extend to only one class but can implement several interfaces.**
10. Generic programming is a facility provided by Java to achieve reusability of code. Generics can be applied to classes and methods. Generic programming is very useful to abstract over types.
11. Java supports single line, multiline, and Java API documentation comments.

12. Byte code is an instruction set of 1 byte (8 bits). Byte codes are compact numeric codes and references (normally numeric addresses).
13. JVM is not portable but byte code is fully portable. JVM is specific to hardware and Operating System.
14. Package is a collection of all class files.
15. Path command is used to link all resources situated in any directory.
16. Classpath is used to inform the Java runtime environment where to find class files.
17. Javadoc is a compiler used to generate API documents for a package.
18. `Java.io.*` is an import statement used to import all files related to `io`. Similarly `java.lang.*` is a package imported to include Java features. This is default package Java imports automatically by default.
19. Keyword `public` means that methods inside a public class can be called outside the class hierarchy.
20. Keyword `static` means that the class can be invoked directly without using an instance of the class to invoke the method `"main"`.
21. Java extended package called Javax houses several graphical libraries which would greatly enhance the graphics components.

## Exercise Questions

### Objective Questions

1. Java has been developed by

1. Gosling and Patric Naughton
2. Bjarne Stroustrup
3. Rambaugh
4. Grady Booch

## 2. Java is

1. Hardware independent
2. Software independent
3. Language independent
4. Firmware independent

## 3. ----- command converts Java source code into byte code class file

1. Java
2. Javac
3. Appletviewer
4. Javadoc

## 4. ----- command loads and executes a Java class file

1. Java
2. Javac
3. Appletviewer
4. Javadoc

## 5. Javadoc compiler produces a file in format of type

1. Byte code
2. HTML format
3. HTTP
4. Word

## 6. Java virtual machine (JVM)

1. Converts Source Code to Object Code
2. Converts Source Code to Byte Code
3. Converts Byte Code to Object Code
4. None of the above

## 7. JVM uses----- to convert a Java byte code class to produce machine-dependent object code

1. Interpreter
2. Compiler
3. Interpreter and JIT
4. a, b and c

## 8. Inheritance is

1. Has-type of relation
2. Is-type relation
3. As-is-type relation
4. None of the above

9. Which of the following statements are true in case of abstract classes?

1. Must contain at least one abstract method
  2. They can be instantiated
  3. Keyword abstract must be present in the class declaration
  4. Class having abstract methods need to be declared as abstract class
- 
1. i
  2. ii
  3. i, iii and iv
  4. ii, iii and iv

10. In command line arguments statement, the arguments are separated by

1. Commas
2. Dot (.)
3. Space
4. Colon

11. `BufferedReader` reads

1. Char by char
2. String per line
3. Word by word
4. Till "Enter" key

#### Short-answer Questions

12. Explain WWW and internet.
13. What are the popular protocols supported by the world wide web?
14. Java is a language for network. Justify.
15. What are byte codes?
16. What is Java Virtual Machine?
17. How do byte code and virtual machine ensure portability?
18. What is interpreter?



19. How do interpreter and JIT compiler improve performance of Java?
20. List out access specifiers of Java.
21. What are the types of inheritances supported by Java?
22. What is the difference between a class and an interface in Java?
23. Explain the terms JDK and API.
24. What is Java package?

#### **Long-answer Questions**

25. Discuss how Java can be truly called language designed for networks.
26. Justify "WORA", write once and reuse it anywhere, philosophy of Java developers.
27. Explain the objective-oriented features of Java.
28. Discuss the similarities and differences between abstract class and interface.
29. Explain the byte codes and JVM role in Java run-time environment.
30. Discuss the architecture of JVM.
31. What are environment variables? Why do we need to set these variables?

#### **Assignment Questions**

32. Explain how Java is most suited for networking program.
33. How does `main()` work in Java environment? Explain the keyword used in `void main()` statement and their relevance.
34. What are objective orientation features program of Java?
35. Explain with a flowchart how a Java program is executed.
36. What are the differences and similarities between C++ and Java?
37. Distinguish between abstract class and interface.

## Solutions to Objective Questions

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

# 16

## Java Fundamentals and Control Loops

### LEARNING OBJECTIVES

*At the end of this chapter, you should be able to*

- Write Java programs with good grasp and understanding of fundamentals.
- Understand data types and their usage.
- Use compound statements and increment and decrement operators.
- Understand operators and their precedence and association rules.
- Use relational and equality operators.
- Understand and use control loops.

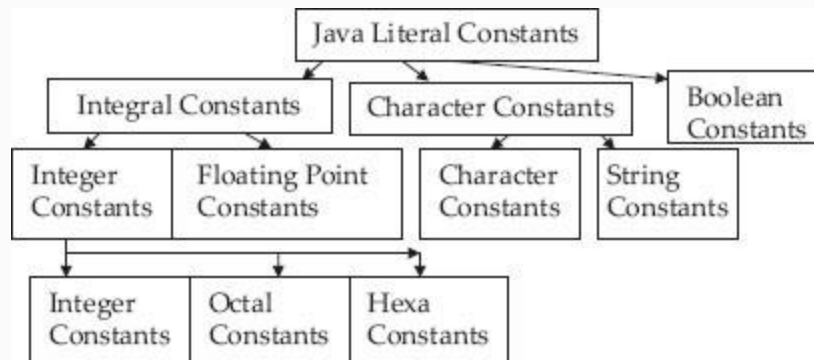
## 16.1 Introduction

In this chapter, we introduce you to Java programming fundamentals like the data types permitted by Java language, together with various operators like logical operators, arithmetic operators, etc. In Chapter 15, you have of course used some of these features, but in this chapter we will provide you with underlying syntax, grammar and theory of the Java language. We have shown working of various types of operators such as binary operators and unary operators and bit-wise operators with sample programs. The precedence and association of operators are presented.

There are basically two types of statements, viz., sequential and control statements. Control statements alter the sequence of execution of statements. Repetitive execution of a block of statements is controlled by counter controlled types of statements executed by commands like for, while, do-while and switch statements.

## 16.2 Constants/Literal Constants

They do not change their value during running of the program. Note that constants are also called literal constants and are presented in Figure 16.1.



**Figure 16.1** Java literal constants

### *16.2.1 Integer Constants*

They can be subdivided into

1. **Decimal integer constants:** 0 10 -745 999

Example: `int myAge=56; // 56 is in decimal`

2. **Octal integer constants:** Only digits between 0 and 7 are allowed. All octal numbers must start with 0 digit to identify as octal number.  
Allowed octal constants: 0777, 001, 0116, 07565L (octal long)

Illegal octal constants are: 089 – 8 is illegal, 777 – does not start with 0  
: -0675.76 – . is illegal

Example: `int myAgeOctal=070; // 70 is octal equivalent of 56`

- 3. Hexadecimal constants:** A hexadecimal number must start with 0x or 0X followed by digits 0 to 9 or alphabets a to f, both uppercase or lowercase allowed. Allowed hexadecimal constants are: 0xffff, 0xa11f, 0x65000UL  
Illegal hexadecimal constants are: 0x14.55, illegal character “.”

Example: `int myAgeHexa=0x38; // 38 is Hexa equivalent of 56`

### *16.2.2 Floating Point Constants/Real Constants*

They are base – 10 number that can be represented either in exponent form or decimal point representation.

Valid floating point constants are: 0.01, 789.89765, 5E-5, 1.768E+9

Invalid floating point declarations are:  
6 invalid . must contain exponent or float point.

5E+12.5 Invalid as exponent cannot be float.  
6,789.00 Invalid character “,”

Example: float area = 567.89; // float value  
double area2 = 567.89; //double value  
double area3 = 5.6789e2 ; // scientific notation

### 16.2.3 Character Constants

Character constants can be declared based on the character set followed in a computer. ANSI have standardized these values as shown below.

A	65	a	097 NULL	000
B	66	b	098 LF(line feed)	010
Z	90	z	122 CR(carriage return)	013

In addition, Java allows Unicode characters like ISO Latin for example: \u0042,\u0043 etc. for Latin character set a, b, etc. A character constant contains a single character enclosed within a pair of single quote marks. Examples of character constants are: '5' 'X' ';' ' '

Note that the character constant '5' is not the same as the number 5. The last constant is a blank space. Character constants have integer values known as ASCII values. Special characters that cannot be printed

normally, double quote (`"`), apostrophe (`'`), question mark (`?`) and backslash (`\`) can be represented by using *escape sequences*. An *escape sequence always starts with \ followed by special character stated above.*

### 16.2.4 String Constants

String constants can contain any number of characters in sequence but enclosed in double quotation marks.

---

```
"new delhi" , "14 Nov 1954" , an empty  
string is "".
```

---

Please note that NULL character `\0` indicates NULL character and is used by Java language to indicate the end of a string.

### 16.2.5 Backlash Character Strings

A few of the backlash character constants that are supported by Java and extensively used in output programming are shown in Table 16.1.



**Table 16.1** Precedence and association rules for the operators

Special Character	Escape Sequence
Bell	\a
Back space	\b
Horizontal tab	\t
Vertical tab	\v
Form feed	\f
New line	\n
Carriage return	\r
Double Quote	\"
Apostrophe/Single quote	\'
Backslash	\\
Null	\0
Octal number	\On
Hexadecimal number	\cHn

### *16.2.6 Boolean Literals*

Boolean data type represents either true or false. For example: `boolean flag =`

`true; boolean ready = false.` Note that we cannot use 1 or 0 to represent true or false as done in C++. Further, observe that true and false are **NOT** enclosed in quotes.

### 16.2.7 Symbolic Constants

Many a times, we need to use symbolic constants such as PI, to represent 3.141519 that remains constant through out the program. A symbolic name substitutes a sequence of characters or a numerical value that follows it. Symbolic names are written in capital letters as a convention. We use declaration *final* and its syntax and examples are shown below:

---

```
final data_type symbolic_name = value ;  
final double PI = 3.14159; final int MAX  
= 50;
```

---

## 16.3 Variables and Assignment of Values to Variables

A variable can consist of alphabets and digits. Either upper- or lowercase or mixtures of both cases are allowed. A

variable cannot start with a digit. It can start with an `_`. The allowable characters in Java language are alphabets A to Z, a to z, numbers 0 to 9. Examples of valid variables are:

---

```
int x;      char a,b,c;      byte b;
double pi;
```

---

Once variables are declared, they need to be assigned values prior to using them. This can be achieved by

- Assigning values to variables. We can assign the values at the time of declaration of variables or just before they are used. Examples are:
- 

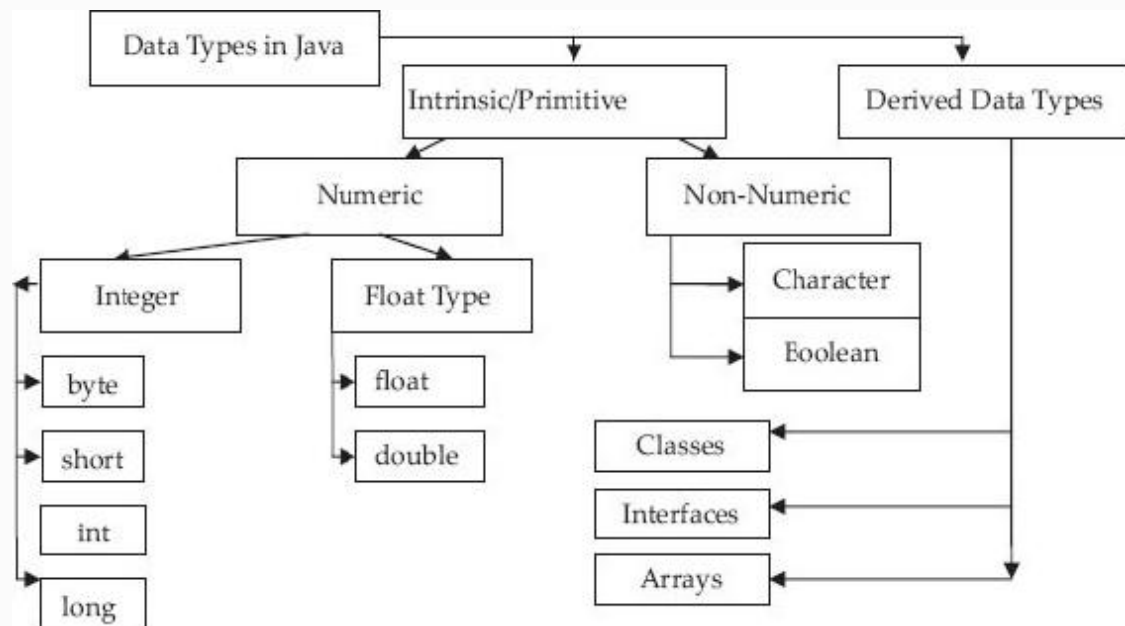
```
int count = 100;
byte m = n = k = 75;
double x = 3.14159;
int x , y = 10; // initializes y to
10
```

---

- Read statement: We can obtain the values interactively from the user from key board using `readLine()` command. In the example that follows, we show how to take input from keyboard in detail in [Chapter 17](#).

## 16.4 Data Types

Data types define the range of permitted values and operations that can be performed on the data type. Data types, also called intrinsic data types or primitive data types, supported by Java language is given in Figure 16.2. The ranges allowed for a 32-bit IBM PC and memory requirements are highlighted in Table 16.2.



**Figure 16.2** Data types in Java

**Table 16.2** Integer data types and their permissible ranges

Type	Size	Min Permissible Value	Max Permissible Value
byte	One Byte	-128	127
short	Two Bytes	-32768	32767
int	Four Bytes	-2,147,483,648	2,147,483,647
long	Eight Bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Data types can also be distinguished as

- Intrinsic or basic data types like int, char, float, double, etc. Intrinsic or basic data types are those that do not contain any other data types.
- Derived data types classes, arrays, interfaces, etc.

### *16.4.1 Integer Data Types*

Java supports four integer data types. The permissible ranges are defined in Table 16.2. Note that it is better to use integer data types as per requirement of storage space rather than routinely declaring the data type as int. Since int takes 4 bytes, it takes longer processing time than byte which is only 1 byte. Also note that Java does not support unsigned data types. All data types are signed, i.e. positive or negative. Long integer constant can be specified by appending the letter `l` s at the end . For example, `789654234L` or `7896s`

### 16.4.2 Floating Point Data Types

Float data types are used to represent fractional part of data such as 2.434. There are two types of floating point numbers viz., single precision and double precision. Float data types and their ranges are presented in Table 16.3. By default Java treats float data type with double precision numbers and to get float, we must either type cast or attach 'f' or 'F' to denote single precision. For example, 1.4397f or 1.4397e4F. Double precision numbers are used when we need higher precision. In Java all mathematical calculations are carried out in double precision only.

**Table 16.3** Floating point data types and their permissible ranges

Type	Size	Min Permissible Value	Max Permissible Value
float	Four Bytes	-3.4e+038	3.4e+038
double	Eight Bytes	-1.7e+308	1.7e+307

### 16.4.3 Character Type

Java supports char data type and it takes 2 bytes but it can store only single characters. Two bytes are allocated because Java has to

cater for larger number of characters in Unicode.

### *16.4.4 Boolean Data Type*

Boolean data type uses only a single bit of storage. It is used to demote true or false condition. The true and false cannot be put in single or double quotes and neither can they be used as identifiers.

#### **Example 16.1: FindMaxDemo.java To Find Program to Find Max of Three Numbers**

```
//FindMaxDemo.java
1. package com.oops.chap16;
2. import java.io.*;
3. class FindMax{
4. public int Max3Numbers(int a ,int b,
   int c){
5. int max=a;
6. if ( max<b) max = b;
7. else if ( max<c) max=c;
8. return max;}
```

```

9. }//end of Findmax
10. public class FindMaxDemo{
11. public static void main(String[]
args) {
12. // create an object of FindMax
13. FindMax obj=new FindMax();
14. int num1=0;int num2=0;int num3=0;int
max=0;
15. try{
16. BufferedReader input = new
BufferedReader (new InputStreamReader
(System.in));
17. System.out.println("Enter value for
number num1:");
18. num1 =
Integer.parseInt(input.readLine());
19. System.out.println("Enter value for
number num1:");
20. num2 =
Integer.parseInt(input.readLine());
21. System.out.println("Enter value for
number num1:");
22. num3 =
Integer.parseInt(input.readLine());
23. }catch(Exception e){}
24. //get data for 3 numbers. Call
GetNumbers() ofFindmax
25. max=obj.Max3Numbers(num1,num2,num3);
26. System.out.println("Maximum of three
numbers:" + num1+" : "+num2 +" : "+ num3
+ " = "+max);
27. }

```



```
28. }// end of class FindMaxDemo
```

**Output:** Enter value for number num1: 45

Enter value for number num1: 67

Enter value for number num1: 12

Maximum of three numbers:45 : 67 : 12 =

67

**L  
i  
n  
e  
N  
o  
.  
3  
:**

is package statement: package  
com.oops.chap16 our class files are stored  
here.

**L  
i  
n  
e  
N  
o  
.  
2  
:**

imports package java.io.\* which is required  
to handle input and output methods.

**L**

declares a class FindMax. This will be class that

**i  
n  
e  
N  
o  
.  
3  
:**

will perform our core function of finding out maximum of three numbers.

**L  
i  
n  
e  
N  
o  
s  
.  
4  
-  
8  
:**

defines a method called `Max3Numbers` that receive three integers `num1`, `num2`, `num3` and finds maximum and returns max number at line no 8.

**L  
i  
n  
e  
N  
o  
.  
1  
0  
:**

is our main class, hence it is declared along with **`public static void`** `main(String[] args)` { statement at Line No 11.

**L  
i  
n  
e  
N  
o  
.  
1  
3  
:**

defines an object of class FindMax FindMax  
obj=new FindMax();

**L  
i  
n  
e  
N  
o  
.  
1  
4  
:**

defines three variables num1, num2, num3  
and max and initializes with 0;

**L  
i  
n  
e  
N  
o  
s  
.  
1  
5**

are part of try and catch block, which is used to  
catch any IO-related errors or exceptions. Note  
that at line no 15 we have included try block  
because at line no 16, we are reading  
BufferedReader input from keyboard.

&  
2  
3  
:

L  
i  
n  
e  
N  
o  
.  
1  
6  
:

```
BufferedReader input = new  
BufferedReader (new  
InputStreamReader(System.in)); It  
shows that System.in is keyboard ,  
standard input device. This has been  
attached to InputStreamReader and read  
into system by BufferedReader object.
```

L  
i  
n  
e  
N  
o  
.  
1  
8  
:

```
num1 =  
Integer.parseInt(input.readLine());  
This is how we will read data from keyboard.
```

You will see more about input statements and classes in succeeding chapters. For now,

get used to our modus operandi of solving problems with objects.

## 16.5 Scope and Life Time of Variables

The class variables and instance variables, i.e. objects of a class, are declared inside a class. Class declaration is enclosed in a set of brace brackets `{ }`. Variables belong to all the instances of the class, the instances are particular to the object being created. We define two characteristics for variables viz. scope and life.

Java is a block-oriented language. Block can be defined as a set of statements enclosed in a pair of controlling brace brackets. Scope is defined as availability across methods function and program segments. Life is defined as life of controlling brace brackets. It also means life of variables extends up to controlling block of statement. Hence these variables are also called local variables, i.e. local to block.

The scope of the variable is local. This means that a variable declared in a function is accessible only within the function.

Further, the life of variables declared within the function is the life of the function itself, i.e. within the brace brackets of the function. Hence these variables are called local variables.

### **Example 16.2: SwapDemo.java A Java Program to Demonstrate Static Variable Usage**

```
//SwapDemo.java to demonstrate local
variables
1. package com.oops.chap16;
2. class Swap{
3. public void SwapVars(int x , int y){
4. int temp;
5. temp=x;x=y;y=temp;
6. System.out.print("X&Y vars inside
Swap Method after swapping");
7. System.out.println(" X = " +x + " Y =
" + y);}
8. }
9. class SwapDemo {
10. public static void main(String[]
args) {
```

```

11. int x=10; int y=100;
12. System.out.print("X&Y variables
before calling Swap Method");
13. System.out.println(" X = " +x + " Y
= " + y);
14. //create an object
15. Swap obj=new Swap();
16. obj.SwapVars(x, y);
17. System.out.print("X&Y variables
after return from Swap Method");
18. System.out.println(" X = " +x + " Y
= " + y);
19. }
20. }// end of SwapDemo

```

Output: X&Y variables before calling  
Swap Method X = 10 Y = 100

X&Y variables inside Swap Method after  
swapping X = 100 Y = 10

X&Y variables after return from Swap  
Method X = 10 Y = 100

**L  
i  
n  
e  
N  
o  
.**

declares a class SwapDemo. In line No 11, we have declared two variables x and y to hold values 10 and 100, respectively.

**9  
:**

**L  
i  
n  
e  
N  
o  
.  
1  
5  
:**

an object is created for class Swap.

**L  
i  
n  
e  
N  
o  
.  
1  
6  
:**

calls a method `SwapVars ( x, y)` a method belonging to Swap class. Variables x and y are passed by call by value method. In this method, x and y are copied to stack area belonging to `SwapVars ()` stack area.

**L  
i  
n  
e  
N  
o  
s**

define a method `SwapVars (int x, int y)`



•  
**3**  
**t**  
**o**  
**9**  
**:**

**L**  
**i**  
**n**  
**e**  
**N**  
**o**  
**•**  
**4**  
**:**

declares a local variable called temp. Even x and y which are copied by calling method `main()` into `SwapVars()` area are local variables.

**L**  
**i**  
**n**  
**e**  
**N**  
**o**  
**•**  
**5**  
**:**

swaps the values of variable x and y using a third variable temp;

**L**  
**i**  
**n**  
**e**  
**N**

display changed variable x and y as 100 and 10 inside `SwapVars()` method as expected.

o  
s  
.  
6  
&  
7  
:

L in `main()` method displays x and y after  
i return from `SwapVars()` method and the  
n result is that variables have not been swapped,  
e with x and y showing originally assigned values  
N 10 and 100. So what has gone wrong? Nothing.  
o This is what call by value and local variables are  
s expected to do. The change made in  
.  
1 `SwapVars()` method on local variable, in this  
6 case temp x and y, are not reflected to calling  
& method `main()`. Why? Because they belong to  
1 different areas of the stack.  
8  
:

Swapping variables without using a third variable: In the method `SwapVars()` of Example 16.2, we have used a third variable called temp to swap the variable. Can we achieve the same result without using the

third variable? Refer to the code segment shown below:

---

```
1. public void SwapVars(int x , int y){  
  // suppose x=10 & y=100  
2. x=x+y; // x now becomes 110  
3. y=x-y; // y now becomes 110-100 =10  
4. x=x-y; // y now becomes 110-10 = 100  
5. System.out.print("X&Y vars inside  
   Swap Method after swapping");  
6. System.out.println(" X = " +x + " Y =  
   " + y);}   
7. }
```

---

This code is more efficient as we have not used a third variable.

## 16.6 Arithmetic Operators

The basic (also known as intrinsic) operators are

---

```
+ addition - subtraction *  
multiplication  
/ division % modulus (remainder after  
division)
```

---

These operators are called binary operators because they operate on two operands. For example,  $a + b$  involves a binary operator  $+$  and two operands  $a$  and  $b$ . In Java, division implies integer division. For example,  $5/4$  would result 1. The  $\%$  operator also called modulus operator would give remainder as result. For example,  $20\%4$  would result in 0 and  $21\%4$  would result in 5.

## 16.7 Type Conversion and Type Casting

### *16.7.1 Type Conversion*

If the variables involved in an operation are of different types, then Java carries out type conversion automatically before the operation. If the operation is between a float and double, the float will be converted to double and the result will result in double. We can say either widening or narrowing takes place in type conversion. The process of conversion to a larger data type from a smaller data type is called widening and conversion from larger to smaller data type is called narrowing. Note that narrowing will

cause truncation of data. If an expression holds byte, short or int, then the result is widened to int. Similarly for floating point, Java automatically converts to double and finally the result is shown as per the largest data type in the expression or as desired by the user through type casting, which is explained in the next section.

### *16.7.2 Type Cast*

Suppose that we want to declare the result in particular data type, we can type cast as shown below. Casting is often used when a method returns a different data type.

The syntax is as follows:

---

```
type var1 = ( type casted) var2;  
int a , b; float x;  
x=(float)a/b; // a/b is integer  
division and the result is converted to  
float.
```

---

**Example 16.3: QuadraticDemo.java  
To Find the Toots of Quadratic**

# Equation

//QuadraticDemo.java

---

```
1. package com.oops.chap16;
2. import java.io.*;
3. class Quadratic{
4. public void FindRoots(int a ,int b,
int c){
5. int d;
6. float root1,root2;
7. System.out.println("The Equation is
");
8. System.out.println( a + "X^2" + "+" +
b+ "*" + "X" + "+" + c);
9. d = ((b*b)-(4*a*c));
10. if (d<0)
11. { System.out.println( "No real
solution since d<0");
12. System.out.println( "roots are
imaginary\n");
13. double sqrd= (double)Math.sqrt(-
d/(2*a));
14. double real = (double) (-b)/(double)
(2*a);
15. System.out.println( "root1 : "+ real
+ "+ i "+ (float) (sqrd));
16. System.out.println( "root1 : "+ real
+ "- i "+ (float) (sqrd));
```

```

17. }
18. if(d==0)
19. { System.out.println( "roots are
real\n");
20. System.out.println( "root1&2 are
equal "+ (float)-b/(2*a));
21. }
22. else
23. if ( d>0)
24. { double sqrd = Math.sqrt(d);
25. System.out.println( "root1 : "+
(float)(-b + sqrd)/(2*a));
26. System.out.println( "root2 : "+
(float)(-b - sqrd)/(2*a));
27. }
28. }//end of FindRoots
29. }//end of Quadratic
30. public class QuadraticDemo{
31. public static void main(String[]
args) {
32. // create an object of Quadratic
33. Quadratic obj=new Quadratic();
34. int num1=0 ;int num2=0; int num3=0;
35. try{
36. BufferedReader input =
newBufferedReader (new InputStreamReader
(System.in));
37. System.out.println("Enter Non-zero
Integer value for number num1:");
38. num1 =
Integer.parseInt(input.readLine());
39. System.out.println("Enter Non-Zero

```

```

Integer value for number num1:");
40. num2 =
Integer.parseInt(input.readLine());
41. System.out.println("Enter Non-Zero
Integer value for number num1:");
42. num3 =
Integer.parseInt(input.readLine());
43. }catch(Exception e){}
44. // Call FindRoots of Quadratic
45. obj.FindRoots(num1,num2,num3);
46. System.out.println("End of Programme
...");
47. }
48. }// end of class QuadraticDemo

```

Output: Enter Non-Zero Integer value for  
number num1:3

Enter Non-Zero Integer value for  
number num1:2

Enter Non-Zero Integer value for number  
num1:-5

The Equation is :  $3X^2+2X+-5$

root1 : 1.0

root2 : -1.66666666

End of Programme ...

Enter Non-Zero Integer value for number  
num1:2

Enter Non-Zero Integer value for  
number num1:3

Enter Non-Zero Integer value for number  
num1:4

The Equation is  $2X^2+3X+4$

No real solution since  $d<0$



```

roots are imaginary
root1 : -0.75+ i 2.236068
root1 : -0.75- i 2.236068
End of Programme ...

```

---

<b>Line No. 3:</b>	declares a class Quadratic, which in turn defines a public method called FindRoots that receives three integer arguments a, b, c.
<b>Line No. 9:</b>	calculates the value of d given by $b^2 - 4 * a * c$ ;
<b>Lines.</b>	deal for $d < 0$ in which case the roots are imaginary and takes the form $-b/2a + i (d/2a)$ . We had resorted to type casting to float or double as a,b,c are integers. The type casting is shown in line No 9: <code>double real = (double) (-b) / (double) (2*a) ;</code>

1  
0  
-  
1  
6:

L  
i  
n  
e  
N  
o  
s.  
1  
8  
-  
2  
1:

deal with a case for  $d = 0$  and roots are equal :  
(float)  $-b / (2*a)$

L  
i  
n  
e  
N  
o  
s.  
2  
2  
-  
2  
7:

deal with a case for  $d > 0$  and roots are given  
by: root1 :+ (float)  $(-b +$   
sqrd) /  $(2*a)$  and root2 : (float)  $(-b$   
- sqrd) /  $(2*a)$  ;

<b>L i n e N o. 3 o :</b>	declares main public class <code>QuadraticDemo</code> and at Line No 33 creates an object for class <code>quadratic</code> .
<b>L i n e N o s. 3 7 t o o 4 3:</b>	obtain values for three coefficients <code>num1</code> , <code>num2</code> , <code>num3</code> form keyboard. Notice that we have used try and catch blocks for detecting any errors while reading input from keyboard using <code>System.in</code> and <code>BufferedReader</code> object.

## 16.8 Unary Operators

In unary operators, operator precedes a single operand. Unary operators are: unary minus ( `-` ), Increment and decrement

operators : ++ , -- . . Examples are : -  
4.0 , -5 \* (A+B)

---

++ i , i++ , --i , i-

---

Minus operator is both a unary and a binary operator. Unary minus, for example -5.0 operates on variable on its right, while the binary operator has variables on both sides of operator and it is an arithmetic operator.

### *16.8.1 Increment and Decrement Operators*

If they precede the operand, the variable is incremented at first and then the operation is performed. If they follow the operand, then the operation is performed first and then the variable is incremented.

**Example 16.4: IncrDecr.java To Demonstrate Pre- and Post-increment Operators**

---

```
//16.4 IncrDecr.java to demonstrate pre
& Post Increment operators
package com.oops.chap16;
public class IncrDecr {
    public static void main(String[] args)
    {
        int count = 1;
        System.out.println("Count = " +
count); // out put will be 1
        /* count will be incremented by one
and then operation of
        print is performed . Output will be
2*/
        System.out.println("Count Pre Incr = "
+ (++count));
        /* Now , if you use count will be
printed first. Output is 2.
        Then count will be incremented by 1 to
3.*/
        System.out.println("Count Post Incr;
but increment will be after statement =
" + (count++));
        System.out.println("Count Post Incr
after executing post incr = " +
(count));
    }
}
```

Output: Count = 1

Count Pre Incr = 2

Count Post Incr; but increment will be  
after statement = 2

```
Count Post Incr after executing post  
incr = 3
```

---

### *16.8.2 Assignment Operator*

The assignment operator is `=`. This is also called equality operator. For example, `int x = 45;` creates a variable of data type `int` and assigns a value 45 to it.

There is a compound assignment operator wherein two operators are combined into one statement. For example, `x += 10;` means `x = x + 10`.

### *16.8.3 Chained Assignment*

The assignment operator together with compound assignment operators can be used in chained assignment statement. For example: `int x = y = z = 0;` this statement is equivalent to statement `x = ( y = ( z = 0 ) )`.

Note that assignment always takes place from right to left. The first assignment is `z = 0`; its reference is passed to `y`, so that `y` is also equated to 0, and then lastly `x = 0`.

### 16.8.4 Relational Operators

The relational operators are : > >= < and <= .

These four relational operators have the same precedence. However, they have lower priority than arithmetic operators. The two more operators, known as equality operators = and !=, have priority just below relational operators.

### 16.9 Logical Operators

These are && and || and NOT operators (!). Evaluation of expressions connected by logical operators are done from left to right and evaluation stops when either truth or falsehood is established. In the statement shown below:

---

```
if ( (iflag==0) && (
youflag= 0) )
```

---

first iflag == 0 is evaluated, if it is true, then only the second expression (youflag= 0) is evaluated. In other

words, evaluation stops as soon as truth or falsehood is established.

We will show the use of logical operators with an example. If attendance is  $> 75$  and sessionals  $\geq 50$ , then set eligible to true, else set it to false. Depending on the eligibility issue, admit card for end term examinations.

### Example 16.5: LogicalOpDemo.java To Show the Usage of Logical Operators Not

```
//LogicalDemo.java
1. package com.oops.chap16;
2. import java.io.*;
3. class Logical{
4. public boolean Eligibility(int marks,
int att){
5. boolean yesno=true;
6. if ( (marks>50)&& (att>=75) )
yesno=true;
7. else yesno=false;
8. return yesno;
```



```

9. }//end of FindReverse
10. }//end of Reverse class
11. public class LogicalDemo{
12. public static void main(String[]
args) {
13. //create an object of Logical
14. Logical obj=new Logical();
15. int marks=0;int attendance=0;String
name="";boolean yesno;
16. try{
17. BufferedReader input = new
BufferedReader (new InputStreamReader
(System.in));
18. System.out.println("Enter Name of
Student");
19. name=input.readLine();
20. System.out.println("Enter marks ");
21. marks =
Integer.parseInt(input.readLine());
22. System.out.println("Enter Attendance
");
23. attendance =
Integer.parseInt(input.readLine());
24. }catch(Exception e){}
25. //Call Eligibility() of class
Logical
26.
yesno=obj.Eligibility(marks,attendance);
27. System.out.println("Name :" + name
);
28. System.out.println("Marks :
Attendance : " + marks + " : "+

```

```

attendance);
29. System.out.println("eligibility:" +
yesno );
30. if( yesno==true)
31. System.out.println(" Collect Admit
Card");
32. else System.out.println("Sorry.
falling short of academic or Attendance
Standard");
33. }
34. }// end of class LogicalDemo

```

#### **Output:**

```

Enter Name of Student Suresh Enter marks
65 Enter Attendance 66
Name: Suresh Marks : Attendance : 65 :
66 eligibility:false
Sorry. falling short of academic or
Attendance Standard
2 run: Enter Name of Student Gautam
Enter marks 98 Enter Attendance89
Name: Gautam Marks : Attendance : 98 :
89eligibility:true
Collect Admit Card

```

<b>L i n e</b>	<b>If</b> (( marks>=50) && (att >=75)) is a statement that contains logical and &&. First the left-hand side (LHS) of &&, i.e. expression
----------------------------	---

```
(marks>=50) , is evaluated. Only if it is true,  
N the right-hand side (RHS) of && expression  
O (att >=75) is evaluated. Only if both are true,  
• only yesno is set to true. Else it is set to false.  
6  
:
```

Observe that && is utilizing the system very efficiently since it evaluates the RHS if and only if the LHS is true.

## 16.10 Bit-wise Operators

There are seven bit-wise operators available in Java language. They are:

**& Bit-wise AND.** Used for masking operation. For example, if you want to mask the first four bits of a number 'n', then we will mask n with a number whose last four bits are 1s, i.e. 0001111. In Octal representation, it is 016. (Remember that an octal number starts with 0 and a hexa number starts with 0x.)

---

```
n = 1 0 0 1 0 1 0 1 =  
149(decimal)  
& 0 0 0 0 1 1 1 1 = 016(octal)  
result n = 0 0 0 0 0 1 1 1
```

---

Note that the last four bits are 0101 and are unaffected, i.e. they are just reproduced in the result, whereas the left four bits are all 0s, i.e. they are *masked*.

| **Bit-wise OR**. This operator is used when you want to set a bit. For example, if we want to set 0th and 2nd bit to 1 for  $n = 144$ , then we will use | operator with  $n$ , as shown below:

---

```
n = 1 0 0 1 0 0 0 0 =  
144(decimal)  
| = 0 0 0 0 0 1 0 1 =  
005(octal)  
result n = 1 0 0 1 0 1 0 1 =  
149(decimal)
```

---

^ **Bit-wise Exclusive OR**. Exclusive OR, also known as odd function, produces output 1 when both bits are not the same

(odd) and produces a 0 when both bits are the same.

---

```
n = 1 0 0 1 0 1 0 1 =  
149 (decimal)  
^ = 0 0 0 0 0 1 0 1 =  
005 (octal)  
result n = 1 0 0 1 0 0 0 0 =  
149 (decimal)
```

---

**<< Left Shift.** Shifting left by one position, bits of a binary number is equal to multiplying the number with 2.

---

```
n = 1 0 0 1 0 0 0 0 =  
144 (decimal)  
n<<1 1 0 0 1 0 0 0 0 0 =  
288 (decimal)  
n<<2 1 0 0 1 0 0 0 0 0 0 = 576
```

---

**>> Right Shift.** Shifting right by one position, bits of a binary number is equal to division of the given number with 2.

---

```
n = 1 0 0 1 0 0 0 0 =  
144 (decimal)  
n>> 0 1 0 0 1 0 0 0 =
```

72(decimal)

$n \gg 2$  0 0 1 0 0 1 0 0 = 36(decimal)

---

### **>>> Bit-wise zero Fill Shift**

**Operator.** This operator shifts right by specified positions but fill the shifted slots with zeros. For example, (  $n \gg 3$  ) would result in

---

$n = 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1 =$

149(decimal)

$n \gg 3\ n = 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0 =$

18(decimal)

---

### **~ Tilde operator. one's**

**complement Operator.** This is a unary operator used to find one's complement of a given number.

---

$n = 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1 =$

149(decimal)

$\sim n = 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0 =$  bit-

wise complement

---

## Example 16.6: Bit-wise.java To Find Whether a Given Number is Prime or Not

```
//BitWise.java to show the concepts of
bit wise operators of java;
package com.oops.chap16;
import java.io.*;
public class BitWise {
public static void main(String[] args) {
int n = 149;
int res;
res = n & 0016;
System.out.println("The resultant of Bit
wise AND operator is :"+ res);
res = n | 0016;
System.out.println("The resultant of Bit
wise OR operator is :"+res);
System.out.println("The resultant of
Logical OR operator is :"+ res);
res = n ^ 0016;
System.out.println("The resultant of
Exclusive operator is :"+ res);
res = n <<2;
System.out.println("The resultant of
shift left ( by 2 bits) operator is :"+
```

```
res);  
res = n >>2;  
System.out.println("The resultant of  
shift right ( by 2 bits) operator is :"  
+ res);  
res = n >>>3;  
System.out.println("The resultant of  
shift right>>> ( by 3bits) operator is  
:" + res);  
res = ~n;  
System.out.println("The resultant of NOT  
operator is :" + res);  
}  
}
```

OUTPUT :

The resultant of Bit wise AND operator  
is :5  
The resultant of Bit wise OR operator is  
:159  
The resultant of Logical OR operator is  
:159  
The resultant of Exclusive operator is  
:154  
The resultant of shift left ( by 2 bits)  
operator is :596  
The resultant of shift right ( by 2  
bits) operator is :37  
The resultant of shift right>>> ( by  
3bits) operator is :18  
The resultant of NOT operator is :-150

---



## 16.11 Other Operators

### *16.11.1 Question Mark (?) Operator Conditional Expressions*

Suppose you want to allot 10 additional bonus marks to students who put in 100 percent attendance, and all others additional 2 marks. This would result in statements like

---

```
If ( attendance > 100)
marks += 10;//this is a compound
statement. It means Marks=Marks +10
else
    marks +=2;
```

---

Java language gives you the facility of conditional operator, using which the above 4 lines can be coded as a single line

---

```
marks = (attendance > 100) ? marks +
10 : marks + 2;
```

---

**The syntax is :  $z = \text{exp1} ? \text{expr 2} : \text{exp3}$ .** Exp1 is evaluated first. If it is true z is equated to the result of exp2 . else z is equated to exp3.

## Example 16.7: Ternary.java To Find Out Maximum of Two Numbers and Three Numbers

```
1. package com.oops.chap16;
2. public class Ternary {
3.     public static void main(String[]
args) {
4.         byte a=56, b=101, c=45, max;
5.         // find the maximum of a & b
6.         max=(a>b)?a:b;
7.         System.out.println("maximum of a &
b =" + max);
8.         max=( ( a>( max=(b>c)?b:c) )?a :max
);
9.         System.out.println("maximum of a &
b & c =" + max);
10.    }
11. }
maximum of a & b =101
maximum of a & b & c =101
```

### 16.11.2 Member Operator or Dot Operator

It is used to denote a package or a member method or a class. For example, we have shown package as: `package com.oops.chap16`. We have further used member methods as: `obj. FindPrime()` etc.

### *16.11.3 Instanceof Operator*

We can use instanceof operator to test if the object belongs to a particular class or not. For example, `boolean yesno; yesno = std instanceof Student` checks if the object `std` is an instance of class `Student` and returns true.

### *16.11.4 New Operator*

It is used to allocate resources when we create objects for classes. New operator allocates resources in heap memory at run-time dynamically. For example:

```
Student std = new Student  
("Ramesh" , 50595, );
```

will allocate resource to object `std` on heap memory.

```
int[] myArray = new int[]  
{10,13,18,20};
```

 will create an array called myArray of data type integer on heap memory and initialize the array with values of 10,13,18,20

### *16.11.5 Operator Precedence and Associativity*

Like in mathematics, in Java too the use of parentheses overrides the basic precedence of arithmetic operators. There are two priorities associated with arithmetic operators: Higher Priority : \* / and % Lower Priority : + - . Therefore, when an expression does not contain any parentheses, Java runs two passes. In the first pass, the higher priorities are considered and in the second pass, the lower priorities are considered.

**Example 16.8: Evaluate the Expression :  $a+b/3*c/2-4$  for  $a=2$ ,  $b=3$ , and  $c=4$**

The given expression is :  $a + b / 3 * c / 2 - 4$   
 $= 2 + 3/3 * 4 / 2 - 4$

First pass: High Priority from left to right

Step 1:  $2 + 1 * 4/2 - 4$  (  $3/3$  is evaluated)

Step 2:  $2 + 4 / 2 - 4$  (  $1*4$  is evaluated) Step 3  
:  $2 + 2 - 4$  (  $4/2$  is evaluated)

Second pass low priority from left to right:  $2$   
 $+ 2 - 4$  (  $2+2$ )

$4 - 4 = 0$  (  $4 - 4$  is evaluated last)

**Example 16.9: Evaluate the  
Expression :  $a+b/(3*c)/(8-4)$  a for  
 $a=2, b=24, c=1$**

The given expression is :  $a + b / (3 * c) / (8 - 4)$  . The expressions is parentheses are highest priority . Hence :  $2 + 24 / (3) / (4)$   
Brackets are evaluated first  
 $2 + 8/4$  left to right :  $2 + 2 = 4$

The association refers to the execution of operators by the compiler. It can be from right to left or from left to right. The precedence and association of operators are summarized in Table 16.4. The operators at the top have priority more than those that appear later in the table, i.e. operator priority is highest at the top of the table and lowest at the bottom of the table. Operators on the same line have the same priority.

**Table 16.4** Precedence and association rules for the operators

Operator	Association Priority
----------	----------------------

Dot operator . Method call ( ) Array ref [ ]	Left to right 1
Logical negation (!) , Tilde( ~), Increment ( ++ ) , Decrement ( -- ) + Unary minus ( - ) , (type) casting	Righ t to left 2
* / %	Left to right 3
+ -	Left to right 4
<< >> >>>	Left to right 5
< <= > >= instanceof	Left to right 6
= = !=	Left to

	right 7
& bit wise &	Left to right 8
^ bit wise XOR	
bit wise or	Left to right 9
&& logical and	
logical or	Left to right 10
	Left to right 11
	Left to right 12
?: conditional operator	Righ t to left 13
= Assignment operator	
	Righ t to left 14



= += -= *= /= %= ^= != <<= >>=	Right to left
,	Left to right

- \* / and % have all the same priority
- Unary operators like + , - , and \* have more priorities than binary operators

## 16.12 Conditional and Branching Statements

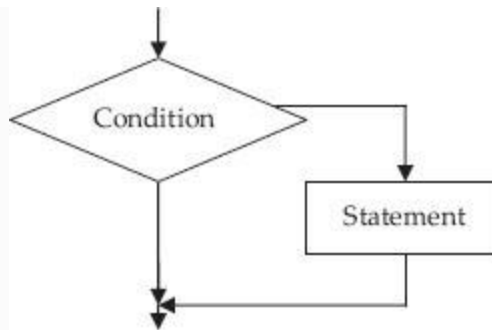
Normally the instructions are executed sequentially. But logic demands that the next instruction to be executed need not necessarily be the next in line, but can be branched to any other statement as per logic of the algorithms. These are called branching statements. The branching can either be conditional or unconditional.

Java is a block-oriented language. The program consists of statements. Statements are logically grouped into blocks. Blocks are enclosed in brace brackets. { and } are used to denote start and end of the block in Java.

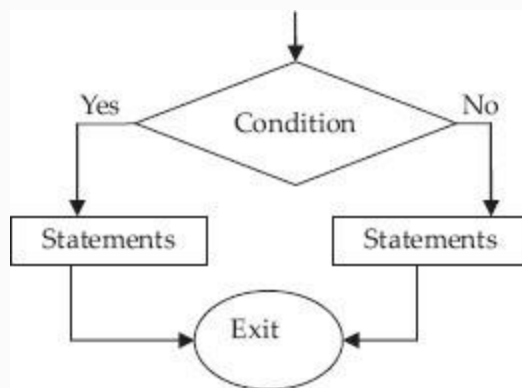
All variables declared inside the controlling braces are called local to the block. It means the value the variable holds is available only inside the block. Statements inside the block are also called compound statements. Note that there will be no semicolon after closing brace brackets. For example, in function definition we have enclosed all the statements in brace brackets.

### *16.12.1 If and If–Else Statements*

Refer to Figures 16.3 and 16.4. Control flow and syntax are shown for if statements and also for if-else statements. Else statement is optional. But if used, it will be associated with the nearest if statement. In the example shown, else is attached to the innermost if.



**Figure 16.3** Control flow in If statement



**Figure 16.4** Control flow in If-else statement

---

```
if ( totalMarks > 60)
if ( total Marks > 70)
System.out.println ("passed with
distinction");
```

```
else System.out.println ("passed with  
First class");
```

---

Use of brace brackets dictate the association rule for else statement. Else in the following code is linked up with the immediate if statement above it

---

```
if ( totalMarks > 60)  
{ if ( total Marks > 70)  
  { System.out.println ("passed with  
distinction");}  
  else // associated with inner if  
  { cout<<"passed with first  
class"<<endl;}
```

---

```
}  
else // associated with outer if  
{ ...else block.....}
```

---

**Example 16.10: IfLoop.java To Test the if Control Loop**

---

//IfLoop.java to test the if loop

```
1. package com.oops.chap16;
2. public class IfLoop {
3. public static void main(String[]
args) {
4. //Declare two arrays to hold marks
and attendance of 5 students
5. int[] idNo = new int[]{ 50595,
50596,50597,50598,50599};
6. float[] marks = new float[]
{96.0F,65.8F,71.0F,46.0F,33.0F};
7. float[] att = new float[]
{78.0F,66.0F,71.0F,75.0F,45.0F};
8. boolean [] yesno = new boolean[5];
9. for ( int i=0;
i<5;i++)yesno[i]=false; //initial value
for yesno
10. // To be eligible attendance >=60.0
and marks>=50.0
11. for (int i=0; i<5;i++)
12. if ((marks[i]>=50.0)&&
(att[i]>=60.0)) yesno[i]=true;
13. System.out.println("List of eligible
candidates");
14. for (int i=0; i<5; i++)
15. if( yesno[i]== true)
16. System.out.println(" IdNo : " +
idNo[i] + " marks :" + marks[i] + "
attendance " + att[i]);
17. }
```

```
18. }//end of IfLoop class
```

**Output:** List of eligible candidates

```
IdNo : 50595 marks :96.0 attendance 78.0
```

```
IdNo : 50596 marks :65.8 attendance 66.0
```

```
IdNo : 50597 marks :71.0 attendance 71.0
```

---

<b>L i n e N o s. 5 t o 7 :</b>	declares three arrays namely <code>idNo[]</code> , <code>marks[]</code> and <code>att[]</code> to denote and initializes the values to the array. When the array is large , we can also use for loop to enter the values as we have done at line No 9 for initializing the array <code>yesno[5]</code> .
---	---

<b>L i n e N o . 1</b>	checks if <code>marks[i] &gt;=50.0</code> and <code>att[i]&gt;=60.0</code> by using a logical and <code>&amp;&amp;</code> operator and sets corresponding <code>yesno[i]</code> to true.
--	---

<b>2</b>	
<b>:</b>	
<b>L</b>	ouputs the list of eligible students if is true.
<b>i</b>	
<b>n</b>	
<b>e</b>	
<b>N</b>	
<b>o</b>	
<b>.</b>	
<b>1</b>	
<b>6</b>	
<b>:</b>	

### *16.12.2 Nested If Statements*

Nested if means if statement within an if statement. The problem in Example 16.8 uses nested if statements to implement simple calculator

**Example 16.11: CalcIf.java To Simulate Simple Calculator With +,-,\*,/ Operators Using Nested Ifs**

---

```
1. package com.oops.chap16;
2. import java.io.*;
3. public class CalcIf {
4. public static void main(String[]
args) throws IOException{
5. float a, b, result=0.0F;
6. int choice;
7. BufferedReader input = new
BufferedReader (new InputStream
Reader(System.in));
8. System.out.println("Enter number 1");
9. a =
Float.parseFloat(input.readLine());
10. System.out.println("Enter number
2");
11. b =
Float.parseFloat(input.readLine());
12. System.out.println("Enter choice 1
Addition 2: Subtraction 3:
Multiplication 4 : Division");
13.
choice=Integer.parseInt(input.readLine()
);
14. if( choice == 1) result = a+b;
15. else
16. if( choice == 2) result = a-b;
17. else
18. if( choice == 3) result = a*b;
19. else
20. if (choice == 4) result = a/b;
21. else
```



```
22. System.out.println("Wrong
Operatot");
23. System.out.println("The result : "
+result );
24. }
25. }//end of CalcIf class
```

**Output:** Enter number 1 100  
Enter number 2 20  
Enter choice 1 Addition 2: Subtraction  
3: Multiplication 4 : Division 4  
The result : 5.0

---

### *16.12.3 If–Else–If Ladder*

Line Nos. 14 to 21 depict the if–else–if ladder. It is called ladder because it looks like a ladder. Observe the indentation. Its deep indent and costs in terms of space. Hence the if–else–if ladder can be replaced by the if–else–if, as shown below:

---

```
if( choice == 1) result = a+b;
else if( choice == 2) result = a-b;
else if( choice == 3) result = a*b;
else if (choice == 4) result = a/b;
else System.out.println("Wrong
Operatot");
System.out.println("The result : "
+result );
```

---

### 16.12.4 Switch and Case Statements

The switch statement is similar to the if–else–if ladder we have used in the previous section. The syntax is `switch (var) {`  
`case value1 : statements1 :`  
`break;`

---

```
case value2 : statements2 : break;
default : statements3; // this statement
is optional
}
```

---

Note that `var` has to be integer and `value` has to be an integer constant. Observe also that case label ends with `:` To demonstrate the concepts involved we show next Example 16.9.

**Example 16.12: SwitchCalc.java To Simulate Simple Calculator with +,-,\*,/ Operators Using Switch Statement Nested Ifs**

---

```
package com.oops.chap16;
import java.io.*;
public class SwitchCalc {
public static void main(String[] args)
throws IOException {
float a, b, result=0.0F;
int choice;
BufferedReader input = new
BufferedReader (new InputStreamReader
(System.in));
System.out.println("Enter number 1");
a = Float.parseFloat(input.readLine());
System.out.println("Enter number 2");
b = Float.parseFloat(input.readLine());
System.out.println("Enter choice 1
Addition 2: Subtraction 3:
Multiplication 4 : Division");
choice=Integer.parseInt(input.readLine()
);
switch(choice)
{
case 1: result = a+b;break;
case 2: result = a-b;break;
case 3: result = a*b;break;
case 4: result = a/b;break;
default: System.out.println("Wrong
Operatot");
} //end of switch
System.out.println("The result : "
+result );
}
```

```
//end of SwitchCalc class
```

**Output:** Enter number 1 100

Enter number 2 200

Enter choice 1 Addition 2: Subtraction

3: Multiplication 4 : Division 4

The result : 0.5

---

## 16.13 Control Loops

There will be several occasions when a programmer has to execute a set of instructions repeatedly till a condition is met. In these situations we need control loops. There are several conditions likely while framing these control loops like:

- Programmer is aware of the initial conditions, loop termination conditions and step increments for testing the conditions. We use for loop for this purpose.
- When the programmer is not aware if the control loop executes, then he has to check the condition first and then enter control loop. We use while statement for this case.
- When the programmer wants the control loop to be executed at least once and then check for the condition, we use do while loop.

### *16.13.1 While Loop*

While loop is written by a programmer if he is not sure if the while block will be executed. A condition is checked first. If it is true, the while block is executed. The syntax of while statement is

While (expression) //body of while contains a single line no brace brackets required

---

```
Statement;
While (expression)
{ statement1;
  statement2;
}
while(true) // expression is always
true
{ block of statements;
} // the loop is called forever while
loop
```

---

Let us do an interesting problem to show the use of control loops. We have a number denoted by 989989. We need sum of digits, i.e.  $9+8+9+9+8+9 = 52$  and no. of digits =6. The method and program are shown below:

---

```
Step1 Rem = n% 10 :// rem =9
Step 2 N=n/10 :// n=989989/10 = 98998 (
```

integer division )

We will repeat Step 1 & 2 while n>0

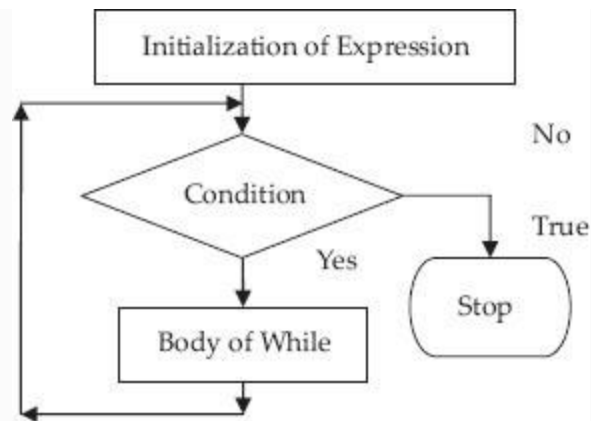
---

## **Example 16.13: Sumdigits.java To Find Program to Find the Sum of Digits in a Number**

---

```
1. package com.oops.chap16;
2. public class SumDigits {
3. public static void main(String[]
args) {
4. int num = 989989,rem,sum=0, count
=0,temp;
5. temp=num; // store it in temp
6. while ( num>0)
7. {rem=num%10; // finds remainder
8. num=num/10;
9. count++; //increments counter
10. sum+=rem; //adds remainder to sum
```

---



**Figure 16.5** Control flow for while and for loops

```

11. }//end of while
12. System.out.println("Given Number" +
temp);
13. System.out.println("No of digits=" +
count);
14. System.out.println("Sum of digits="
+sum);
15. }
16. }//end of class
Output:Given Number989989
No of digits=6
Sum of digits=52

```

<b>Line No. 6:</b>	shows while loop. The condition $n > 0$ is checked first. Only if it is true, the loop is entered.
<b>Line No. 7:</b>	is modulus operator that gives remainder.
<b>Line No. 8:</b>	is an integer division. This means that it ignores the remainder

### 16.13.2 Do-while Loop

The syntax is

```

Do
{
    block of statements
} while (expression);

```

Do while loop is most appropriate when we want the control loop to be executed at least once. The block is executed first and the condition is checked. If true, the loop is executed till the condition becomes true. We will use do-while loop when we know that the loop needs to be executed at least once,



whereas while loop is used when we are not aware if the loop needs to be executed or not. Control flow is shown in Figure 7.6.

**Example 16.14:**  
**SumDigitsDoWhile.java a Program to find sum of digits and also no of digits using do while**

```
1. package com.oops.chap16;
2. public class SumDigitsDoWhile {
3.     public static void main(String[]
args) {
4.         int num = 989989,rem,sum=0, count
=0,temp;
5.         temp=num; // store it in temp
6.         do
7.         {rem=num%10; // finds remainder
8.         num=num/10;
9.         count++; //increments counter
10.        sum+=rem; //adds remainder to sum
11.        }while( num>0);
12.        System.out.println("Given Number" +
temp);
13.        System.out.println("No of digits=" +
```

```
count);  
14. System.out.println("Sum of digits=" +  
sum);  
15. }  
16. }//end of class  
Output: Given Number989989 No of  
digits=6 Sum of digits=52
```

---

### *16.13.3 For Loop*

***For*** loop, as control loop is used, when we know the exact number of times the loop needs to be executed. The syntax of for loop is

---

```
for ( exp1 ; exp2 ; exp3 )  
{ block of statements }  
where exp1 is initialization block  
exp2 is condition test block  
exp3 is alter initial value assigned  
to exp1
```

---

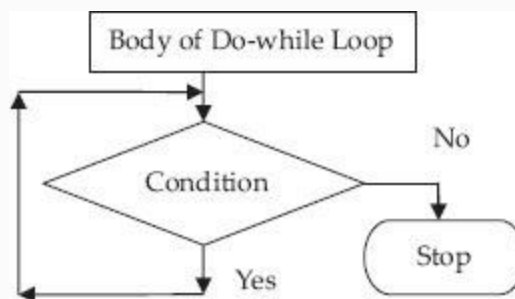
Forever or infinite for loop is shown below. Observe that forever for loop has no initial and final conditions. Note that infinite for loops are run forever. We have to forcibly stop the program by <cntrl - break>

---

```
for ( ; ; )  
{statement;}
```

---

for loops can be nested. That means we can write for loop with in a for loop. In nested for loop, the inner loop is executed for each value of outer loop. For example, the inner loop is executed for  $i = 0$  and the value of  $j$  is varied from 0 final condition. The outer loop is executed for values of  $i$  varying from 0 to  $n - 1$ .



**Figure 16.6** Control flow for Do-while loop

The syntax is

---

```
for ( int i=0;i<n;i++)  
{for (int j=0;j<n;j++)  
    {statement}  
}
```

---

### **Example 16.15: Fibonaacifor.java** **//Program to Find Sum of n** **Numbers and Their Average Using** **For Loop**

---

```
package com.oops.chap16;  
public class Fibonaaci{  
public static void main(String[] args) {  
    int [] FibArray = new int [20];  
    int numOfTerms = 20; //number of terms  
    in the series  
    FibArray[0]=0; FibArray[1]=1; // first  
    two number of the series  
    for (int i =2 ; i< numOfTerms; i++)  
    { FibArray[i]=FibArray[i-1]+FibArray[i-  
    2];}  
    System.out.println(" The fibonaacii  
    series for 20 terms ...");
```

```
for (int i =0; i< numOfTerms; i++){  
    System.out.print(FibArray[i] + " ");  
}  
}  
} //end of class  
Output: The fibonaacii series for 20  
terms ...  
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377  
610 987 1597 2584 4181
```

---

## 16.14 Break

Break statement is used to exit from the switch control or control loop. We can use break statement to exit from for, while and do while, and switch control statements. You have already seen use of break statement in Switch statement. When used within a nested block, a break statement can be used to go to the end of the block in which the break is used. There is another use in break statement. For example: `break label2;` takes you to the end of the block labeled as `label2`.

## Example 16.16: BreakUses.java: A Program that Demonstrates Break Statement

```
1. package com.oops.chap16;
2. import java.io.*;
3. public class BreakUses {
4.     public static void main(String[]
args) throws IOException {
5.         int count=0; int sum=0,num;
6.         // write a forever for loop for
demonstration of
7.         // break for going out of control
loop
8.         BufferedReader input = new
BufferedReader (new InputStreamReader
(System.in));
9.         for(;;){
10.            if ( count ==5)
11.            { System.out.println("upper limit of
5 reached");
12.             break;}
13.         else{
14.             System.out.println("enter value of
"+(count+1)+ "number");
15.             num =
```

```

Integer.parseInt(input.readLine());
16. sum+=num; count++;}
17. }// end of for
18. // Second use of break with label
19. label1:{System.out.println("Entered
label1 ");
20. label2:{System.out.println("Entered
label2 ");
21. label3:{System.out.println("Entered
label3 ");
22. System.out.println("Inside a label3
Using break label2 Statement");
23. break label2;
24. }//end of label3
25. }//end of label2
26. System.out.println("reached end of
label2 ");
27. }//end of label1*/
28. } //end of main()
29. }//end of class

```

**Output:** enter value of 01number10  
enter value of 11number20  
enter value of 21number30  
enter value of 31number40  
enter value of 41number50  
upper limit of 5 reached  
Entered label1  
Entered label2  
Entered label3  
Inside a label3 Using break label2  
Statement  
reached end of label2

<b>Line No. 9:</b>	uses for evr for loop. It can be only stopped by Line No 10 which tests for <code>count == 5</code>
<b>Line No. 11:</b>	shows break statement. If <code>count == 5</code> is true, break statement breaks the for loop. If count is not 5 it enters else loop.
<b>Line Nos . 19 to 27:</b>	show the use of <b><code>break label;</code></b> statement. Each label represents a block of statement. Line No 23 <code>break label2;</code> takes the control to end of <code>label2</code> .

## 16.15 Continue Statement

Continue is used when we want to stop further processing of loop statements and start at the beginning of the control loop. In the example shown below, we would like to



add 10 points to all odd numbers between 0 and 10 and skip adding to even numbers.

### Example 16.17: Continue.java : A Program that Demonstrates Continue Statement

```
1. package com.oops.chap16;
2. import java.io.*;
3. public class Continue {
4.     public static void main(String[]
args){
5.         int count=0;
6.         for(count = 0;count<=10;count++){
7.             if ( count%2==0)
{System.out.println("The number "+count+
" is even ");
8.             continue;} //control goes to beginning
of for loop
9.         System.out.println("The number
"+count+ " is odd");
10.     }//end of for
11. }// end of main
12. }//end of class
```

#### Output:

The number 0 is even The number 1 is odd

The number 2 is even  
The number 3 is odd The number 4 is even  
The number 5 is odd  
The number 6 is even The number 7 is odd  
The number 8 is even  
The number 9 is odd The number 10 is  
even

---

<b>Line No. 7:</b>	checks if <code>(count%2 ==0)</code> i.e. if count is even.
<b>Line No. 8:</b>	continue statement ensures that if the number is even the control goes to the beginning of the loop and does not enter the balance code at line no 9.

## 16.16 Summary

1. Java supports integral, character and Boolean constants. Literal constants are those that do not change their value during the running of the program.
2. Integral constants are Integer constants, Octal constants and Hexa constants.
3. Floating point or real constants are expressed in decimal or exponent form.
4. Character constants are character and String constants.
5. Boolean literals are expressed as true or false.
6. Constants declared as final cannot change their value during running of the program.
7. Integer data types are `byte` (1 byte), `short` (2 bytes), `int` (4 bytes), `long` (8 bytes).
8. Float data types are `float` (4 bytes) and `double` (8 bytes).
9. The scope of the variable is local. This means that a variable declared in a function is accessible only within the function.
10. The basic (also known as *intrinsic*) operators are + addition - subtraction \* multiplication / division % modulus (remainder after division).
11. **Type Conversion:** If the variables involved in an operation are of different types, then Java carries out type conversion automatically before the operation. The process of conversion to a larger data type from a smaller data type is called widening and conversion from larger to smaller data type is called narrowing.
12. **Type Cast:** Suppose we want to declare the result in a particular data type, we can type cast the variable :  
(data type) variable.
13. **Unary Operators:** In unary operators, the operator precedes a single operand. The minus operator is both a unary and a binary operator.

14. ++ , -- operators are called increment and decrement operators.
15. **Assignment Operator** operator is =. This is also called equality operator.
16. The relational operators are : > >= < and <=. Two more operators, known as equality operators == and !=, have priority just below relational operators.
17. **Logical Operators:** These are && and || and NOT operator (!). Evaluation of expressions connected by logical operators are done from left to right and evaluation stops when either truth or falsehood is established.
18. Bit-wise operators available in Java language are Seven. They are & Bit wise AND, | Bitwise OR, ^ Bit wise Exclusive OR, << Left Shift, >> Right Shift, >>> Bitwise zero Fill Shift Operator, ~ Tilde operator . one's complement Operator.
19. **instanceof** operator to test if the object belongs to a particular class or not.
20. **new operator** It is used to allocate resources when we create objects for classes.
21. Syntax of if statement: if (expression) statement.
22. There will be several occasions when a programmer has to execute a set of instructions repeatedly till a condition is met. In these situations, we need control loops. The switch statement is similar to if-else-if ladder.
23. The syntax of while loop is : While (expression) { statement1; statement2; }.
24. Do-while loop :The syntax is Do{ block of statements} while (expression).
25. For loop syntax is: for ( exp1 ; exp2 ; exp3), where exp1 is initialization block, exp2 is condition test block, exp3 is alter initial value assigned to exp1.

26. Break statement is used to exit from the switch control or control loop.
27. Continue is used when we want to stop further processing of loop statements and start at the beginning of the control loop.

## Exercise Questions

### Objective Questions

1. Char data type supported by Java takes

1. 4 bytes
2. 3 bytes
3. 2 bytes
4. 1 byte

2. Float data type supported by Java takes

1. 2 bytes
2. 8 bytes
3. 4 bytes
4. 1 byte

3. Which of the following statements are true in respect of Boolean constants?

1. Enclosed in single quote
2. Enclosed in double quotes
3. Occupies a single bit in memory
4. Can be used as identifiers

1. i
2. i and ii
3. i, ii and iii
4. iii

4. Which of the following statements are true in respect of data types of Java?

1. Byte takes 1 byte
2. Short takes 1 byte

- 3. Integer takes 2 bytes
- 4. Long takes 8 bytes

- 1. iv
- 2. i and ii
- 3. i, ii and iii
- 4. iii

5. Which of the following statements are true in respect of variables of Java?

- 1. Instance variables belong to all instances of class
- 2. Class variables are declared inside or outside the class
- 3. Object is a class variable
- 4. Scope of the variable is local

- 1. ii and iii
- 2. i and ii
- 3. i, ii and iii
- 4. iii and iv

6. Which of the following statements are true in respect of operators of Java?

- 1. Arithmetic operators have the same priority
- 2. \* / % have the same priority
- 3. The priority of + - is higher than the priority of \* /
- 4. Association for arithmetic operators is from left to right

- 1. ii and iv
- 2. i and ii
- 3. i, ii and iii
- 4. iii and iv

7. Modulus operator can only be applied to integer data types TRUE/FALSE

#### Short-answer Questions

- 8. What are the different data types of Java?
- 9. Distinguish a variable and constant with examples.
- 10. Explain type casting with examples.
- 11. What are relational and logical operators? Discuss their priorities.
- 12. What is the difference between & , && operators?

13. What is the difference between = , = = operators?
14. Explain ? operator.
15. Explain new and instanceof operator.
16. Show the working of bit-wise AND operator with examples. What is masking operation?
17. Show the working of bit-wise OR and Exclusive OR operators with examples.
18. Distinguish unary and binary operators.
19. What is >>> operator in Java?

#### Long-answer Questions

20. What are the literal constants and data types provided by Java? Explain with suitable examples.
21. Distinguish type casting and type conversion with examples.
22. Explain the usage of bit-wise operators of Java.
23. What is operator precedence and Associativity? Discuss with examples.
24. Explain the difference between if–else–if and switch statements. Which is better and why?
25. Explain variations in for loop statements in Java with examples.
26. Which one of these are better for control loop: for, while, do while?
27. Distinguish continue and break statement.

#### Assignment Questions

28. Write a c module to compute simple and compound interest using the formula

---


$$SI = P \times N \times R / 100 \text{ and } CI = P \times (1 + R/100)^N$$


---

29. Write a program to store number, age and weight in three separate arrays for 10 students. At the end of data entry, print what has been entered.
30. Using the formula  $A = \text{Squareroot}(s * (s-a) * (s-b) * (s-c))$ , compute area of the triangle.  $S = (a+b+c) / 2$ , and a, b, and c are sides of the triangle
31. Write a Java program to count the number of lines, the number of words, the number of open braces and the number of close braces in an input file.
32. Write a program to convert a given integer into
  1. hexadecimal numbers and
  2. octal numbers.
33. Write a program to shift given integer to 2 positions to the left.
34. Write a program to test if the given integer has 1 in 4 bit position. If it is 0, set it to 1.
35. Write a Java program to mask given integer of the low 4 bits.
36. Using priorities of the operators evaluate  $z = 4 * 5 / 6 + 10 / 5 + 8 - 1 + 7 / 8$ .
37. Write a program to print the ASCII table for range 30 to 122.

### Solutions to Objective Questions

1. c
2. c
3. d
4. a
5. d
6. a
7. False





# 17

## Simple IO and Arrays and Strings Vectors

### LEARNING OBJECTIVES

*At the end of the chapter, you will be able to understand and write programs using*

- Simple IO statements using `BufferedReader` and `InputStreamReader`.
- `Read()`, `readLine()` and `StringTokenizer` class and its methods.
- Java's input using scanner class and output using `String format()`.
- Arrays and Strings handling.
- Vectors and wrapper classes.

## 17.1 Introduction

In Chapters 15 and 16 you have been introduced to some simple IO statements. We continue in this chapter to introduce you to further IO programs as these are required for carrying out meaningful programming in Java. You will learn techniques to input primitive data types using `BufferedReader` and `InputStreamReader` and `read()` and `readline()` commands. `StringTokenizer` which will be used for inputting multiple data in a single line will also be introduced in this chapter.

## 17.2 Input from Keyboard

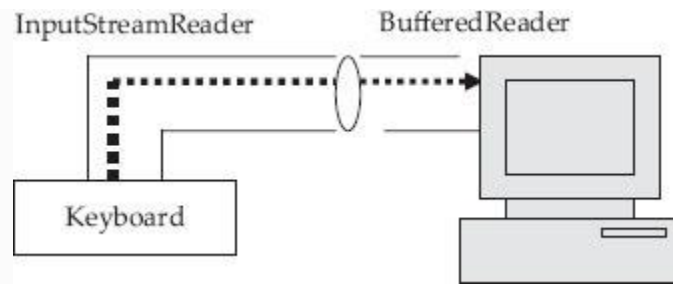
### *17.2.1 System.in, System.out and System.err Commands*

In this section, we will see how to take string and numeric input for Java applications using standard input or output and in Java keyboard is represented by

- **System.in**: Standard input is from keyboard. This represents `InputStream` Object
- **System.out**: Console is represented by `System.out`. This represents `PrintStream` object.
- **System.err**: Java handles stream objects for input and output and hence is likely to encounter errors or exceptions. `System.err` is used to log errors that occur while handling streams. We need to catch these errors and exception objects using try and catch blocks. We will tell you how to use these commands now and reserve more detailed discussion to later chapters on errors and exceptions.

We will use `BufferedReader` to read one String per line. Figure 17.1 shows two streams – `InputStreamReader` and `BufferedReader` – that are connected together. Two streams are required because the computer is a high-speed device and keyboard is a slow device.

`InputStreamReader` accepts the input from keyboard and converts the ascii to data types required by computer and `BufferedReader` stores them so that Java can accept them when they are ready to be picked up.



**Figure 17.1** Input from keyboard

We will read name of the student. We will also read identification number and marks of the student. We will also read a single character from keyboard

### **Example 17.1: Student.java: To Obtain Input String, Int and Float Data from Console**

// Example 17.1 Student.java : To obtain input String , int and double data from keyboard using InputStreamReader

```
1. package com.oops.chap17;  
2. import java.io.*;
```

```

3. import javax.swing.*;
4. public class Student {
5. public static void main(String[]
args) throws IOException{
6. BufferedReader in;
7. in=new BufferedReader(new
InputStreamReader(System.in));
8. System.out.print(„Enter name : „);
9. String name =in.readLine();
10.    System.out.print(“Enter your
Roll Number :”);
11.    String input=in.readLine();
12.    int id=Integer.parseInt(input);
13.    System.out.print(“Enter your
Total Marks : ”);
14.    String input2=in.readLine();
15.    double
totalmarks=Double.parseDouble(input2);
16.    System.out.print(“Enter Grade
of the Student : ”);
17.    char ch = (char)
in.readLine().charAt(0);
JOptionPane.showMessageDialog
Dialog(null,“Name ”+ name +
18.    “\nRoll Number :” + id +”
\nTotal Marks :” +
19.    totalmarks+” \n Grade ” + ch,”
output display”
,JOptionPane.PLAIN_MESSAGE); }
20. }

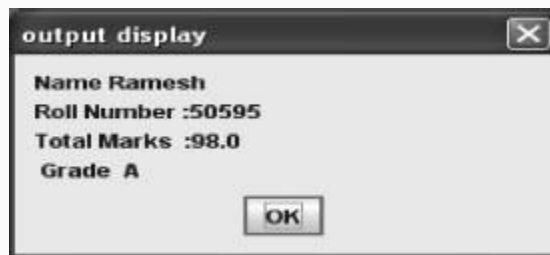
```

### **Output:**

Enter name : Ramesh

Enter your Roll Number :50595  
Enter your Total Marks : 999  
Enter Grade of the Student : A

---



**L  
i  
n  
e  
N  
o  
.  
2  
:**

`import java.io.*;` is required as we are using `InputStreamReader`, `BufferedReader`, and `IOException`

**L  
i  
n  
e  
N  
o  
.**

`void main()` throw `IOException` object. This is an example of method(`void main()`) throwing an exception.  
Observe in Line No. 6: `System.in` is for console input. `InputStreamreader` is attached to `System.in`.

<b>5</b> :	InputStreamReader in turn belongs to BufferedReader.
---------------	---

<b>L</b> <b>i</b> <b>n</b> <b>e</b> <b>N</b> <b>o</b> <b>s</b> <b>.</b> <b>8</b> <b>&amp;</b> <b>9</b> :	show how to accept a string from keyboard : String input=in.readLine();
---	--

<b>L</b> <b>i</b> <b>n</b> <b>e</b> <b>N</b> <b>o</b> <b>s</b> <b>.</b> <b>1</b> <b>0</b> <b>t</b> <b>0</b> <b>1</b> <b>2</b> :	show the commands for accepting integer from keyboards:
---	--

	<pre>10    System.out.print("Enter your Roll Number :"); 11String input=in.readLine(); //Accept the integer as String 12 <b>int</b> id=Integer.parseInt(input); // conver String into integer.</pre>
--	--



**L** show the commands for accepting double from  
**i** keyboards

**n** 13 System.out.print("Enter your Total  
**e** Marks : ");  
**N** 14 String input2=in.readLine();//Accept  
**o** the integer as String  
**s** 15 double  
**.** totalmarks=Double.parseDouble(input2);  
**1**  
**3**  
**t**  
**o**  
**1**  
**5**  
**:**

**L** show the commands for accepting a character  
**i** from keyboard using a readLine() command.

**n** Note that readLine read the String(). charAt()  
**e** returns first character of the string. Hence our  
**N** command would be: char ch = (char)  
**o** in.readLine().charAt(0); We could also  
**s** write this as: String ch =  
**.** in.readLine();  
**1**  
**6**  
**t**  
**o**  
**1**  
**7**  
**:**

### *17.2.2 StringTokenizer to Receive Multiple Inputs in a Single Line*

Those of you familiar with C will know that in scanf statement for receiving input from the keyboard we can read several variables together in a single line. How do we achieve the same result in Java? By using a class called `StringTokenizer` of `java.util` package. Firstly we create an object of `String` class and read the in:

```
String stg = input.readLine(); // to  
read a string from keyboard
```

Assume that our input data is separated by commas. For example, "Ramesh" , 50595, 89.0, "A" representing name, RollNo, total marks and grade.

Make an object of `StringTokenizer` and pass the `String` object as a parameter to `StringTokenizer` object :

```
StringTokenizer stkn = new  
StringTokenizer( stg, ",");
```

StringTokenizer breaks the single line string into tokens based on the separating symbol comma. We can retrieve the token using `nextToken()`. `String tkn = stkn.nextToken();`

We will attempt a problem in which we will read name, number, credits and debits of an employee. Compute net pay and display the result.

## Example

### 17.2: EmployeeCredits.java: To Obtain Multiple Inputs in a Single Line Using StringTokenizer.

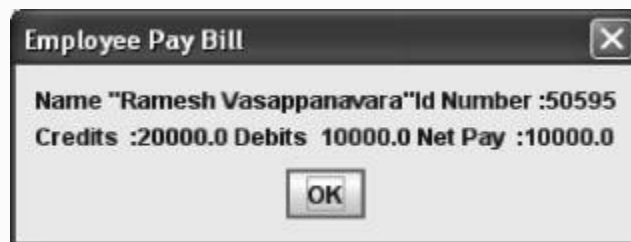
---

```
1. package com.oops.chap17;  
2. import java.util.*; // For using  
StringTokenizer()  
3. import java.io.*;  
4. import javax.swing.JOptionPane;  
// Swing components
```

```
5.  public class EmployeeCredits {
6.  public static void main(String[]
args) throws IOException{
7.  String name,idNo,crdts,debts;//
variables or reading input date
8.  double credits,debits,netPay; //
variables for internal computations
9.  BufferedReader input = new
BufferedReader (new InputStreamReader
(System.in));
10. System.out.println("Enter Emp
name, idNo,credits,debits separated by
comma");
11. String stg = input.readLine(); //
multiple data separated by comma
12. StringTokenizer stkn = new
StringTokenizer(stg,","); // object of
StringTokenizer
13. name = stkn.nextToken(); // break
into tokens
14. idNo = stkn.nextToken();
idNo.trim(); // trim is to remove
leading trailing spaces
15. crdts = stkn.nextToken();
crdts.trim();
16. debts =
stkn.nextToken();debts.trim();
17. // convert to double
18. credits =
Double.parseDouble(crdts);
19. debts =
Double.parseDouble(debts);
```

```
20. netPay = credits-debits;
21.
JOptionPane.showMessageDialog(null,"Name
"+ name +
22. "\tId Number :" + idNo
+"\nCredits :" +
23. credits+"\t Debits " + debits
+"\t Net Pay :"+ netPay,"Employee Pay
Bill",JOptionPane.PLAIN_MESSAGE);
24. }
25. }
```

---



<b>L i n e N o . 2 :</b>	<pre>imports java.util.* a package housing StringTokenizer()</pre>
--	--

**L  
i  
n  
e  
N  
o  
.  
1  
1  
:**

`String stg = input.readLine();` reads String input from keyboard.

**L  
i  
n  
e  
N  
o  
.  
1  
4  
:**

`idNo = stkn.nextToken();`  
`idNo.trim();` shows how to break a continuous string into token by using `nextToken()`. Further `idNo.trim()` removes leading and trailing spaces if any.

**L  
i  
n  
e  
N  
o  
1  
7  
:**

converts String to double data. This is required to do internal computations.

### *17.2.3 Obtaining Inputs Using Java's Scanner Class*

While StringTokenizer class discussed in the previous section does its job, it is cumbersome in that we have to perform several tasks such as creating an object, passing an argument, using the `trim()` and finally to convert into primitive data type. Java has provided a simpler alternative called Scanner class in `java.util` package.

The first action is to create object. Scanner `scn = new Scanner(System.in);`  
`int idNo = scn.nextInt();` is the only statement we need to use to begin the computing process. Similarly, `nextDouble()`, `nextLong()`, `next.Float()` would be used for other data types. Here in Scanner class, when data is inputted from keyboard, variables are required to be separated by spaces. Blank space will be treated as the end of a token.

## Example 17.3: ScannerInput.java: To Obtain Multiple Inputs in a Single Line Using Scanner Class of Java

```
1. package com.oops.chap17;
2. import java.util.*; // for Scanner
class
3. import java.io.*;
4. import javax.swing.JOptionPane;
5. public class ScannerInput {
6. public static void main(String[]
args) throws IOException{
7. String name;// variables or
reading input date
8. int idNo;
9. double credits,debits,netPay;
10. System.out.println("Enter Emp
name, idNo,credits,debits separated by
spaces");
11. Scanner scn = new
Scanner(System.in);
12. name = scn.next();
13. idNo = scn.nextInt();
14. credits = scn.nextDouble();
15. debits = scn.nextDouble();
16. netPay = credits-debits;
17.
```



```

JOptionPane.showMessageDialog(null, "Name
"+ name +
    18. "\tId Number :" + idNo +
\nCredits :" +
    19. credits+" \t Debits " + debits +
\t Net Pay :" +
    20. netPay, " Employee Pay Bill"
,JOptionPane.PLAIN_MESSAGE);
    21. }
    22. }

```

---



<b>Line No. 2:</b>	java.util.* for Scanner class.
<b>Line No. 11:</b>	Scanner scn = new Scanner(System.in); creates an object of Scanner class. System.in represents key board.
<b>Line</b>	shows how to take in String data by using

<b>No. 12:</b>	<code>name=scn.next();</code>
<b>Line No. 13:</b>	shows how to take in int data by using <code>idNo=scn.nextInt();</code>
<b>Line Nos. 14 &amp; 15:</b>	shows how to take in Double data credits & debits by using : <code>credits=scn.nextDouble();</code> <code>debits=scn.nextDouble();</code>

### 17.2.4 Using Control Formats – *System.out.printf()*

You might have used C language `printf()` command wherein we have used controlling formats to display primitive data types. Java also provides such a facility to format and display primitive data types using `System.out.printf()` command. The controlling formats offered by Java are shown in Table 17.1:

**Table 17.1** Formatting for use with `System.out.printf()`

%s : String	%c : char	%d : Decimal Integer
%f : float	%o : octal	% x or % X : hexa
%n : new line	%e or %E : scientific notation	

## Example 17.4: PrintfJava.java: To Send the Formatted Output Using System.out.Printf(

```

1. package com.oops.chap17;
2. import java.util.*;
3. import java.io.*;
4. import javax.swing.JOptionPane;
5. public class PrintfJava {
6. public static void main(String[]
args){
7. String name;// variables or
reading input date
8. int idNo;
9. double credits,debits,netPay;

```

```

10.      System.out.println("Enter Emp
name, idNo,credits,debits separated by
spaces");
11.      Scanner scn = new
Scanner(System.in);
12.      name=scn.next();
13.      idNo=scn.nextInt();
14.      credits =scn.nextDouble();
15.      debits=scn.nextDouble();
16.      netPay=credits-debits;
17.      System.out.println("Employee
pay Bill....");
18.      System.out.printf("String =
name : %s idNo : %d" ,name,idNo );
19.      System.out.printf("String =
Credits:%8.2f debits: %8.2f NetPay
%8.2f" ,credits,debits,netPay );
20.      }
21.  }
22.  Output: Enter Emp name,
idNo,credits,debits separated by spaces
23.  "Ramesh" 50595 20000.00 2000.00
24.  Empleoyee pay Bill....
25.  String = name : "Ramesh" idNo :
50595String = Credits :20000.00 debits :
2000.00 NetPay 17000.00

```

---

<b>Li</b>	shows the usage of <code>System.out.printf()</code>
-----------	---

<b>ne N o. 17:</b>	<code>command. ("String = name : %s idNo : %d" , name, idNo)</code> is passed as an argument.
--------------------------------	---

<b>Li ne N o. 19 :</b>	shws usage of <code>%8.2 f</code> for credits , debits , and <code>netPay</code> fields. It means total formatting space of 8 spaces out of which 2 spaces after decimal.
--	---

### *17.2.5 Formatted Output with String Format*

Java has provided an alternative to format when only strings are involved in outputting we can use `String.format()` method. This is a static method and hence can be called directly using `String.format()` method.

**Example 17.5: StringFormat.java:  
To Send the Formatted Output  
Using String.format()**

---

```
1. package com.oops.chap17;
2. import java.util.*;
3. import java.io.*;
4. public class StringFormat{
5. public static void main(String[]
args){
    6. String name;// variables or
reading input date
    7. int idNo;
    8. double credits,debits,netPay;
    9. System.out.println("Enter Emp
name, idNo,credits,debits separated by
spaces");
    10.    Scanner scn = new
Scanner(System.in);
    11.    name=scn.next();
    12.    idNo=scn.nextInt();
    13.    credits =scn.nextDouble();
    14.    debits=scn.nextDouble();
    15.    netPay=credits-debits;
    16.    System.out.println("Empleoyee
pay Bill....");
    17.    String stg =
String.format("name : %s idNo : %d"
,name,idNo );
    18.    System.out.println(stg);
    19.    stg=String.format("Credits
:%8.2f debits : %8.2f NetPay %8.2f"
,credits,debits,netPay );
    20.    System.out.println(stg);
    21.    }
```

```
22.      }  
Output: Enter Emp name,  
idNo,credits,debits separated by spaces  
"Ramesh" 50595 20000.0 2000.0  
Empleoyee pay Bill....  
name : "Ramesh" idNo : 50595  
Credits :20000.00 debits : 2000.00  
NetPay 17000.00
```

<b>L i n e N o . 1 7 :</b>	<code>stg=String.format("name : %s idNo : %d" ,name,idNo );</code> shows the formatting the argument for name and idNo.Line No. 17 similarly show the formatting for credits and debits and netPay using <code>%8.2f</code>
--	---

## 17.3 Arrays

In day-to-day life, there are several occasions wherein we have to store data of the same type in contiguous locations For

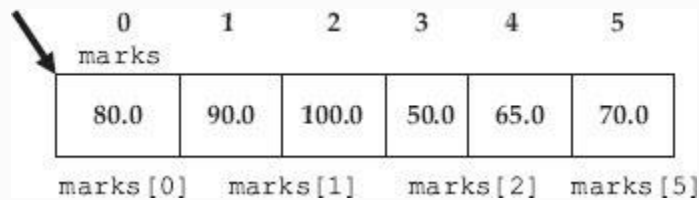
example, consider a situation where you would like to store the marks obtained by a student in six different subjects. Java provides us with a feature called Arrays which can be used to accomplish the above task.

An array is defined as a collection of elements where each element is of the same type. The elements of the array can either be elementary data types like `int`, `float`, `char` or they can be complex data types like structures and objects. An array is a data structure that defines a contiguous memory location for a single data type. It can be a group of students' marks or it can be a group of employees' salaries. For example, `float marks[5];` represents an array with 5 marks. The particular value of an array can be accessed by referring to cell number called index of an array. For example, `marks[5]` refers to marks at index 5.

Generally, the indexes are numbered for 0 to  $n-1$ , where  $n$  is the number of elements in the array. Marks obtained by a student in six different subjects are shown in an array



named ***marks*** in **Figure 17.2**. Elements of the array are referenced by array name followed by subscript. We have shown an array named `marks`; six subject marks scored by the student can be represented by `marks[0]=80.0` and `marks[5]=70.0` etc.



**Figure 17.2** Representation of an array

### *17.3.1 Declaring and Creation of an Array*

Java provides us with two methods to declare arrays. The first method involves a two-step procedure, as follows:

**Step 1:** Declaration of the array using the syntax `datatype [ ] arrayname` for example to declare an array of integers with

the name `integer_array` we would declare as follows:

---

```
int [ ] integer_array
```

---

**Step 2:** Creation of the array using the syntax `arrayname = new datatype[size]`; this step basically assigns storage space in the memory for the array. For example, if we want `integer_array` declared in step1 to accommodate 6 integers we would create as follows:

---

```
integer_array = new int[6]
```

---

Note that while programming both declaration and creation, i.e. both step 1 and step 2 can be combined as follows. Also note that when we declare arrays in such a fashion all the elements in the array are initialized to 0.

---

```
int [ ] integer_array = new int[6]
```

---

The second method is used when you want to declare arrays using some initial values. To declare arrays with some initial values we use the syntax datatype

*[ size(optional)] arrayname = { initializer\_list} . For example:*

---

```
int [6] marks = {40, 50, 60, 70, 80, 90} or int [ ] marks = {40, 50, 60, 70, 80, 90}
```

---

Both the above declarations are valid and we do not have to specify the size of the array. The compiler will calculate the size of the array based on the initializer list.

### *17.3.2 Initialization of Arrays*

- If the array size is small, we can use dynamic declaration and allocation all in a single line as : `int [] myArray = new int[5] { 10, 20, 30, 40, 50};`
  - If the array size is large, we will use for loop to enter the data
- 

```
int [] myArray = new int[size];
String input;
BufferedReader in;
in=new BufferedReader(new
```

```
InputStreamReader(System.in));  
    for ( int i =0; i<size; i++) {  
        System.out.println(" Enter Array [ "  
+ i +" ] element");  
        input=in.readLine();  
        myArray[ i] =  
Integer.parseInt(input);  
    } // end of for loop
```

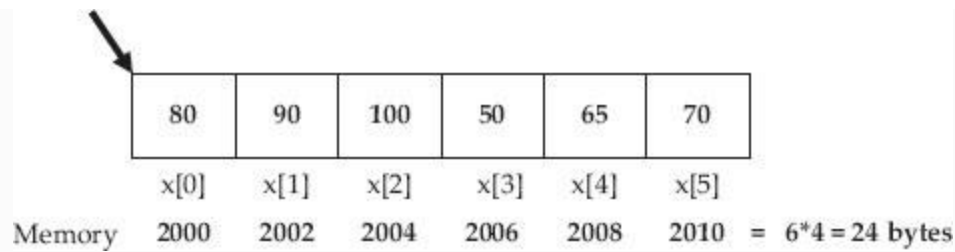
---

### *17.3.3 How Are Arrays Stored in the Memory?*

Consider an array named x, declared as `int`

```
[] x = new int [  
{80, 90, 100, 50, 65, 70};
```

The addresses shown above are dummy addresses. Using of `size of` operator would tell us the memory requirement of data type `int` on your hardware. Assuming that it is 24 bytes, the memories of the array element are shown in [Figure 17.3](#).



**Figure 17.3** Representation arrays with memory locations shown

### *17.3.4 Accessing and Modifying Array Elements*

To access an element in the array we must provide the array name and the index of the element in the array. Array indices start from 0 and go up to array length  $-1$ , where index 0 is for the first element and index length  $-1$  is for the last element. For example, in [Figure 17.3](#), the array marks have a length of 6, so indices will range from 0 to 5. The general syntax for accessing elements of the array is

*arrayname[ expression ]*. It is absolutely essential that the value of the expression must be an integer and the value must be in the range of the array indices, i.e. [0,

length-1] else we will encounter an array out of bounds exception. Let us see a few examples on accessing array elements; all the statements below use the marks array shown in Figure 17.2.

---

```
marks[3] = 50 // statement will change
value stored at index 3 in marks array
to 50
val = marks[2] //statement will store
value present at index 2 (100) into
variable val
int index = 6
marks[index -1] = 70 //expression inside
bracket evaluates to 4 so same as
marks[5] = 75
marks[index +1] = 10 //expression
evaluates to 7, so we get array out of
bound exception
```

---

Java provides us with a length attribute through which we can get the length of the array at run-time. The syntax for getting the length of the array is `arrayname.length`. Usage of length attribute is particularly useful when processing array elements using for loops as demonstrated in the following example.

## Example

### 17.6: TestArrayLength.java To Obtain the Length of the Array

```
1.  package com.oops.chap17;
2.  public class ArrayLengthTest {
3.  public static void main(String[]
args) {
4.      int[] arr = new int[]
{10,20,30,40,50,60,70};
5.      System.out.println(" Given
Array.....");
6.      for(int i = 0; i < arr.length ;
i++ )
7.      { System.out.print(arr[i] + ", "
);}
8.      System.out.println("\nGiven Array
Length :" +arr.length);
9.  }
10. }
```

**Output:** Given Array.....

10, 20, 30, 40, 50, 60, 70,

Given Array Length :7

<b>Line No. 6:</b>	shows the usage of <code>arr.length</code> . Here <code>arr.length</code> returns the length of array <code>arr</code> .
--------------------	--

### *17.3.5 Passing Arrays as Arguments to Methods*

Arrays can be passed as parameters to functions. To define a method with an array as a parameter use the following syntax:

```
ReturnType  
MethodName (datatype[] arrayname) .
```

To pass an array as an argument in the calling function use the following syntax:

```
MethodName (arrayname) ,
```

i.e. use the array name without the brackets. In the program shown below in Example 17.7, when we pass the array to the function



using the call, the array is passed by reference.

### **Example 17.7: SortIntArray.java: To Sort the Integer Array by Sending Array to a Method**

```
1. package com.oops.chap17;
2. class IntSort1{
3. public void Sort(int myArray[],
int len){
4. int temp;
5. for(int i=0; i <len-1; i++){
6. for ( int j=i+1; j<len; j++){
7. if(myArray[i]<myArray[j]){ //swap
8. temp=myArray[i];
myArray[i]=myArray[j]; myArray[j]=temp;
} //end of if
9. } // end of inner for loop
10. } // end of outer for loop
11. } //end of method Sort
12. } //end of class IntSort
13. class IntSortDemo {
14. public static void
main(String[] args) {
15. IntSort1 obj = new
```

```

IntSort1();//make an object
16.    int[] myArray = new int[]
{10,13,17,20};
17.    int len = myArray.length;
18.    System.out.println(" Given
Array .....");
19.    for(int i=0;i<len;i++)
20.    System.out.print( " " +
myArray[i]);
21.    obj.Sort(myArray, len);
22.    System.out.println(" \nSorted
Array .....");
23.    for(int i=0;i<len;i++)
24.    System.out.print( " " +
myArray[i]);
25.    }
26.    }
Output : Given Array ..... 10 13 17 20
Sorted Array ..... 20 17 13 10

```

<b>L</b> <b>i</b> <b>n</b> <b>e</b>  <b>N</b> <b>o</b> <b>.</b>	<p><b>public void Sort(int myArray[], int len){</b> method defines procedure for sorting an array. Observe that <b>int myArray[]</b> is passed as an argument. No need to mention the dimension. Line No. 8 swaps the variable <b>myArray[i]</b> if <b>myArray[i]&lt;myArray[j]</b> as shown at Line No. 7.</p>
--	---

**3**  
**:**

**L** defines an object of `IntSort1`. Line No. 21 calls  
**i** the `Sort()` method of class `IntSort1` by  
**n** passing `myArray` and `length` as arguments.  
**e** Observe that `myArray` is passed as reference.

**N**  
**o**  
**.**  
**1**  
**5**  
**:**

### *17.3.6 Returning Arrays as Arguments to Methods*

We have seen through the above example how arrays can be passed as parameters to a function. Similarly, arrays can also be a return type. To define a method with an array as a return type, we use the following syntax:

```
dataType[] methodName() {  
    function body}
```

Program to demonstrate array as return type of a function. Method copies the input array into a new array and returns the new array.

## Example

### 17.8: ArrayCopyDemo1.java : To Copy an Array and Return the Array to Calling Method

```
package com.oops.chap17;
class ArrayCopy1{
public int[]Copy( int[] myArray) {
    int [] arr = new int[myArray.length];
    for(int i=0; i <myArray.length; i++)
        arr[i]=myArray[i];
    return arr;
} //end of method Copy
} //end of class ArrayCopy
class ArrayCopyDemo1 {
public static void main(String[] args) {
ArrayCopy obj = new ArrayCopy();//make
an object
int[] myArray = new int[]
{10,15,20,30,35};
```

```

int[] copyArray = new
int[myArray.length];
    System.out.println(" Given Array .....");
for(int i=0;i<myArray.length;i++)
    System.out.print( " " + myArray[i]);
copyArray=obj.Copy(myArray);
System.out.println(" \nCopied Array .....");
for(int i=0;i<copyArray.length;i++)
    System.out.print( " " + copyArray[i]);
}
}
Output : Given Array ..... 10,15,20,30,35
Copied Array .....") 10,15,20,30,35

```

---

### *17.3.7 Multi-dimensional Arrays*

Arrays can have more than one dimension. For example, a matrix is a two-dimensional array with a number of rows and a number of columns as show in Figure 17.4.

		Columns			
		0	1	2	3
Rows	0	10			
	1				
	2			20	
	3			17	

This is matrix A with dimensions 4 X 4 written as A[4][4]. First dimension is rows and second dimension is columns.

As per Java convention elements in row major representation is

A[0][0] A[0][1] A[0][2] A[0][3]  
A[1][0] A[1][1] A[1][2] A[1][3]  
A[2][0] A[2][1] A[2][2] A[2][3]  
A[3][0] A[3][1] A[3][2] A[3][3]

Note A[0][0] = 10 ; A[2][2] = 20  
A[3][2] = 17

**Figure 17.4** A two-dimensional matrix

As an example, to demonstrate the two-dimensional arrays we would consider the problem of finding a transpose of a matrix.

### Example 17.9: MatTranspose.java A Program to Find Transpose of a Matrix

```
// Matrix transpose
1. package com.oops.chap19;
2. import java.io.*;
3. class Matrix{
4. void ReadMat(int A[][],int rows,
```

```

int cols){
    5. //input matrix related data using
    scanner
    6. Scanner scn=new
    Scanner(System.in);
    7. for ( int i=0;i<rows;i++)
    8. for ( int j=0;j<cols;j++)
    9. { System.out.print("Enter["+i+" ]"
    +"["+j+" ] element :");
    10.   A[i][j]=scn.nextInt();}
    11.   }//end of ReadMat
    12.   void DisplayMatrix(int A[]
    [],int rows, int cols){
    13.     for ( int i=0;i<rows;i++)
    14.     {for ( int j=0;j<cols;j++)
    15.     { System.out.print( A[i][j] +"
    ");}
    16.     System.out.print( "\n");
    17.     }
    18.     }
    19.   void TranspMat(int A[][],int
    rows, int cols){
    20.     for (int i=0;i<cols;i++)
    21.     {for (int j=0;j<rows;j++)
    22.     { System.out.print( A[j][i] +"
    ");}
    23.     System.out.print( "\n");
    24.     }
    25.   }//end of TranspMat
    26.   }//end of Matrix
    27.   class MatTranspose{
    28.     public static void

```

```

main(String[] args)
    29.    { Scanner scn = new
Scanner(System.in);
    30.    System.out.print("Enter no of
rows & columns:");
    31.    int rows= scn.nextInt();
    32.    int cols= scn.nextInt();
    33.    int A[][]= new int[rows][cols];
//two-dimensional matrix
    34.    Matrix mat = new Matrix();
//create an object
    35.    mat.ReadMat(A,rows,cols);
    36.    System.out.println("Given
Matrix");
    37.    mat.DisplayMatrix(A,rows,cols);
    38.    System.out.println("Transposed
Matrix");
    39.    mat.TranspMat(A, rows, cols);
    40.    }
    41.    }//end of MatTranspose

```

**Output** : Enter no of rows & columns:3

3

Enter[0][0] element :1 Enter[0][1]

element :2 Enter[0][2] element :3

Enter[1][0] element :4 Enter[1][1]

element :5 Enter[1][2] element :6

Enter[2][0] element :7 Enter[2][1]

element :8 Enter[2][2] element :9

Given Matrix

1 2 3

4 5 6

7 8 9



Transposed Matrix

1 4 7

2 5 8

3 6 9

**L  
i  
n  
e  
N  
o  
.  
2  
9  
:**

creates an object for Scanner class called `scn`.  
Line Nos. 31 and 32 reads no of rows and  
columns from keyboard using `nextInt()`

**L  
i  
n  
e  
N  
o  
.  
3  
3  
:**

`int A[][]= new int[rows][cols];`  
defines two-dimensional matrix called `A[][]`.  
Line No. 35 calls `ReadMat()` method and Line  
No. 37 calls `DisplayMatix()` and Line No. 39  
calls `TranspMat()` methods of class `matrix`.  
All these methods pass two-dimensional matrix  
and rows and columns as arguments.

**L** are two for loops for reading matrix elements  
**i** row major wise. Also observe that at line No. 10  
**n** : `A[i][j]=scn.nextInt();` we have read  
**e** element `A[i][j]` using Scanner object as  
**N** `scn.nextInt();`  
**o**  
**s**  
**.**  
**7**  
**&**  
**8**  
**:**

**L** **viz for (int i=0; i<cols; i++) { for (int**  
**i** `j=0; j<rows; j++)` observe that outer loop we  
**n** have used `cols` and inner loop we have used  
**e** `rows`, thus producing the transpose of a matrix.  
**N**  
**o**  
**s**  
**.**  
**2**  
**0**  
**&**  
**2**  
**1**  
**:**

### 17.3.8 Java.util. Arrays Class

Java.util package provides a useful collection class called **Arrays** class. Arrays class defines several static methods that can be used directly without creating an object. A few of the important and useful static methods by Arrays class are discussed here. Note that we will show an example for only one primitive data type but the syntax and concepts are equally applicable to other data types. *Datatype in **Table 17.2*** refers to primitive data type, for example, say int.

**Table 17.2** Arrays class static methods

Arrays class static methods	Functionality
-----------------------------	---------------

```
static datatype  
binarySearch  
datatype  
array[],datatype  
val);
```

Can be applied to sorted arrays. Uses binary search method

```
static boolean  
equals(datatype  
myArray1[],datatype  
myarray2[]);
```

```
static boolean  
deepEquals (Object  
[]  
obj1,Object[]obj2)
```

Returns true if two arrays are equal  
  
Returns true if both arrays and their nested arrays if any are equal

```
static void  
sort(datatype  
myArray[]);
```

```
ststic void  
sort(datatype  
myArray[],int  
start,int end)
```

Sorts the array  
  
Specifies the range in an array for sorting.

```
static void fill(  
datatype myArray[],  
datatype val);
```

Fills all elements of array with val. If index is beyond,

ArrayIndexOutOfBoundsException Exception occurs
--

## Example

### 17.10: ArraysClassDemo.java: To Show Usage of Arrays Class Static Methods

```
1. package com.oops.chap17;
2. import java.io.*;
3. import java.util.Arrays;
4. public class ArraysClassDemo {
5.     public static void main(String []
args)
6.         throws
IOException,ArrayIndexOutOfBoundsException{
7.         int[]myArray = new int[12];
8.         // fill the array with zeros as
initial values
9.         Arrays.fill(myArray, 0);
10.        System.out.print("\nArray with
initial values\n");
11.        DisplayArray(myArray);
```

```

12.    System.out.print("\nnow
allocate values to array");
13.    //populata the array in aloop
14.    for ( int i=0;i<=
(myArray.length)/2;i++)
15.        myArray[i]=(4*i);
16.    for ( int i=myArray.length-
1;i>myArray.length/2;i--)
17.        myArray[i]=i;
18.    System.out.print("\nArray after
allocating values\n");
19.    DisplayArray(myArray);
20.    System.out.print("\nSorting the
array...");
21.    Arrays.sort(myArray);
22.    System.out.print("\nArray after
sorting values\n");
23.    DisplayArray(myArray);
24.    System.out.print("\nBinary
Search the array...");
25.    int pos =
Arrays.binarySearch(myArray,4);
26.    System.out.print("\n index = " +
(pos+1));
27.    }//end of main
28.    static void DisplayArray(int
myArray[]){
29.        for ( int
i=0;i<myArray.length;i++)
30.            System.out.print(myArray[i]+"
");
31.        // System.out.println("\n");

```

```

32.    }// end of DisplayArray
33.    }// end of ArraysClassDemo}
end of ArraysClassDemo

```

**Output:**

```

Array with initial values
0 0 0 0 0 0 0 0 0 0 0 0
now allocate values to array
Array after allocating values
0 4 8 12 16 20 24 7 8 9 10 11
Sorting the array...
Array after sorting values
0 4 7 8 8 9 10 11 12 16 20 24
Binary Search the array...
Pos = 2

```

<b>Line No . 3:</b>	<code>imports java.util.Arrays.</code>
<b>Line No . 6:</b>	<code>throws IOException,ArrayIndexOutOfBoundsException.</code>
<b>Line</b>	<code>Arrays.fill(myArray, 0);</code> uses static method of Arrays called <code>fill(array[],arg)</code> to fill the array with all

<b>No . 9:</b>	zeros. Note that a static method can be called directly with class name. No need to invoke them with object as we normally do with class objects.
<b>Lin e No s. 11 an d 19 an d 23:</b>	<code>DisplayArray (myArray) ;</code> I a static method displayed in the same class. Hence we have used the specifier static.
<b>Lin e No s. 14 to 17:</b>	populate the arrays with values <code>(4*i)</code> and I in a loop.
<b>Lin e No . 21:</b>	calls <code>Arrays.sort (myArray)</code> to sort the array



## 17.4 String

A String is an array of characters or a sequence of characters. In Java, the String occupies an important position and role because Java is a language created for network and to transmit data on the network we have to use the Strings. In C or C++, a string is an array of character, with the last one being '\0'. In Java, this is not so. A String is an object in Java. They are implemented by **String class** and **StringBuffer class**. **We can use any one of the following three methods to create a String Object:**

**Method 1 :**     `String name ; name =  
new String("Hello Class");`

Or we can combine both statements into one

---

```
String name = new  
String("Hello Class");
```

---

**Method 2 :** convert character arrays into Strings.

---

```
char x[] = {'G','o','o','d'};
String stg = new String
(stg);
```

---

**Method 3 :** String name =" Hello  
Class";

There are two important methods with  
Strings that are widely used.

---

```
int len = name.length() ; //
gives the length of String name
```

---

Strings can be concatenated as shown below:

---

```
System.out.println ( name + " "
+ x); // gives out Hello Class Good
```

---

### *17.4.1 Array of Strings*

We can create an array of Strings. For  
example, we want to create an array of  
Strings containing 6 city names.

---

```
//create array of Strings
String [] cityName= new String[6];
Scanner scn = new Scanner(System.in);
```

```
int i=0;
int len = cityNames.length;
System.out.println("Enter 6 city
names separated by spaces");
for ( i=0;i<len;i++)
cityNames[i]=scn.next();
```

---

Example solved at Ex 17.6 will show the working of Array of strings. We will obtain the strings into an array called `cityNames` using Scanner methods and sort the array of strings alphabetically.

### *17.4.2 String Class Methods*

Method supported by string class is shown in Table 17.3.

**Table 17.3** String class methods

String Method	Functionality Performed
<pre> stg.length();  stg.CharAt(pos) ;  stg1.compare(stg2);  stg1.concat(stg2); </pre>	<p>Returns length.</p> <p>Returns char at pos.</p> <p>Returns -1/0/1 on stg1&lt; or = or &gt; stg2</p> <p>Concatenates stg1 &amp; stg2</p>
<pre> stg2=stg1.toLowerCase();  stg2=stg1.toUpperCase();  stg2=stg1.replace('a', 'b');  stg2=stg1.trim(); </pre>	<p>Converts stg1 to lower case</p> <p>Converts stg1 to upper case</p> <p>Replaces char a with char b</p> <p>Removes leading &amp; Trailing white space of stg1.</p>
<pre> stg1.equals(stg2); </pre>	<p>Returns true if stg1 equals stg2</p> <p>Returns true if stg1 equals stg2 ignoring the case</p>

<code>stg1.equalsIgnoreCase();</code>	
<code>stg1.substring(pos);</code>  <code>stg2.substring(start,end);</code>  <code>string.valueOf(s);</code>  <code>stg.toString();</code>  <code>stg.indexOf('x');</code>  <code>stg.indexOf('x',pos);</code>	<p>Returns substring from pos</p> <p>Returns substring specified by start and end-1.</p> <p>Converts s into an object of String.</p> <p>Converts object stg to string</p> <p>Returns position of first occurrence of x.</p> <p>Returns position of first occurrence of x after pos.</p>

## Example

### 17.11: StringMethodsDemo.java: To Show Usage of String Class Methods

---

```
1. package com.oops.chap17;
2. public class StringSort {
3. public static void main(String[]
args) {
4. //create array of Strings
5. String [] cityName= new String[]
{"MADRAS" ," BANGALORE" ," GHAZIABAD" ,"
BINTULU" ," VIZAG" ," RAIPUR" };
6. // Get the size
7. int len = cityName.length;
8. System.out.println("\nCity Names
.....");
9. for ( int i=0;i<len-1;i++)
10.     System.out.print(cityNames[i]+"
");
11.     String temp;
12.     for ( int i=0;i<len-1;i++){
13.         for ( int j=i+1;j<len;j++){
14.             if(
cityNames[j].compareTo(cityNames[i])<0)
15.                 { // swap city names
16.                     temp=cityNames[i];
cityNames[i]=cityNames[j];
17.                     cityName[j]=temp;
18.                 }
19.             }//innerfor loop
20.         }//outer for loop
21.     System.out.println("\nCity
Names Sorted alphabetically");
22.     for ( int i=0;i<len-1;i++)
23.         System.out.print(cityNames[i]+"
```

```
");
24.    }//end of main
25.    }// end of class StringSort
Output: City Names .....
MADRAS BANGALORE GHAZIABAD BINTULU
VIZAG
City Names Sorted alphabetically
BANGALORE BINTULU GHAZIABAD MADRAS
RAIPUR
```

---

**L  
i  
n  
e  
N  
o  
.  
6  
:**

creates a String Object called `cityNames[]` and allocates resources using new operator.

**L  
i  
n  
e  
N  
o  
.**

gets length of String `cityNames`. Observe that length has no brackets.

7  
:

**L** are outer & inner for loops for sorting the  
**i** String Array cityName[].  
**n**  
**e**  
**N**  
**o**  
**s**  
**.**  
**1**  
**2**  
**&**  
**1**  
**3**  
**:**

**L** **if**(cityNames[j].compareTo(cityNames[  
**i** i])<0)uses compareTo() method. This  
**n** method compares cityName[j]&  
**e** cityName[i]) and returns <0 or 0  
**N** or >0 i.e negative, zero, and  
**o** positive depending on of  
**.** cityName[i]) is alphabetically  
**1** appears before cityName[j]  
**4**  
**:**

### 17.4.3 StringBuffer Class



String is a fixed length and modifying the length and content is not allowed, i.e., they cannot be mutated (they are immutable).

`StringBuffer` class is meant to fulfill the gap in requirements of programmers. We can insert in the middle or at the end of a string. Hence `StringBuffer` class elements are mutable. Methods supported by `StringBuffer` class are shown in Table 17.4.

**Table 17.4** `StringBuffer` class methods

StringBuffer Class Methods	Functionality
<pre>stg.setCharAt(pos, 'c');</pre>	<p>Changes character at pos to c.</p>
<pre>stg1.append(stg2);</pre>	<p>Appends stg2 to stg1.</p>
<pre>stg1.insert(pos, stg2);</pre>	<p>Inserts stg2 at pos.</p>
<pre>stg.setLength(n);</pre>	<p>Sets length of stg to n. Truncation or addition with zeros takes place if original length is &gt; n or &lt; n.</p>

## Example

### 17.12: StringBufferDemo.java: To Show Usage of StringBuffer Class Methods

---

```
1. package com.oops.chap17;
2. import java.util.*;
3. public class StringBufferDemo {
4. public static void main(String[]
args) {
5. //create object of StringBuffer
6. StringBuffer stg = new
StringBuffer("C++ & Java");
7. System.out.println("\nGiven String
: " + stg);
8. int len = stg.length();
9. System.out.println("Printing Given
String character by character");
10. for (int i=0;i<len;i++)
11.
System.out.print(stg.charAt(i)+" ");
12. System.out.println("\nInserting
a string :OOPS in: at the beginning ");
13. String stg2=" OOPS in ";
14. stg.insert(0,stg2);
15. System.out.println("String
after inserting");
16. System.out.println(stg);
17. System.out.println("Modifying
few elements of String");
18. stg.setCharAt(0,'o');
19. stg.setCharAt(1,'o');
20. stg.setCharAt(2,'p');
21. stg.setCharAt(3,'s');
22. System.out.println("String
after modifications");
```

```

23.    System.out.println(stg);
24.    System.out.println("Appending
String");
25.    stg.append(" Languages");
26.    System.out.println("String
after appending");
27.    System.out.println(stg);
28.    }//end of main
29.    }//end of class

```

StringBufferDemo

**Output:** Given String : C++ & Java  
Printing Given String character by character

C + + & J a v a

Inserting a string :OOPS in: at the beginning

String after inserting

OOPS inC++ & Java

Modifying few elements of String

String after modifications

oops inC++ & Java

Appending String

String after appending

oops inC++ & Java Languages

<b>Line</b>	StringBuffer stg = new StringBuffer("C++ & Java"); creates
-------------	---

<b>No. 3:</b>	an object of <code>stringBuffer</code> class and assigns initial value as ("C++ & Java");
<b>Line No. 11:</b>	prints the string chara by character using : <code>charAt()</code> method.
<b>Line No. 14:</b>	inserts the string "OOPS in " using : <code>insert()</code> method.
<b>Line No. 17 to 21:</b>	modifies the character using <code>setCharAt()</code> method.
<b>Line No. 25:</b>	append a string "Languages" to the existing String using <code>append()</code> method

### *17.4.4 StringBuilder Class*

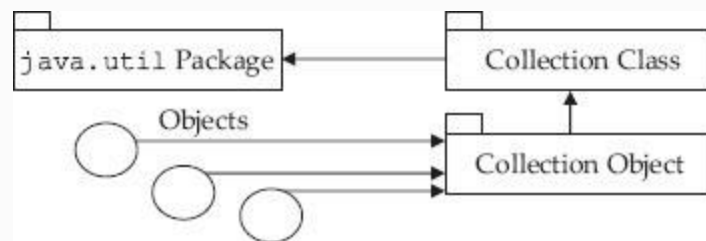
Jdk1.5 has provided `StringBuilder` class in addition to `StringBuffer` class.

`StringBuilder` class is the same as that of `StringBuffer`, but it is not synchronized. What this means is that when several threads are waiting as `StringBuffer` class is synchronized, all the waiting threads will be executed as per priority one after the other and thereby prevent a condition called racing. More details about synchronized methods and racing are provided in the chapter on multithread programming. When programming with single thread, `StringBuilder` will provide quicker solution but multithreaded programming use of `StringBuffer` is recommended.

## 17.5 Collection Framework

Arrays, while they are extremely useful, have their own limitations. They are static in nature. This means that resources are to be allocated at compile time. Arrays are fixed length data structures. Further arrays contain only homogenous data types. How do we store a group of objects in an array? J2SE 5.0 has provided a collection

framework. This is somewhat similar to container of C++ but has quite a number of differences. Collection framework is implemented in `java.util` package. The concept is: make a collection object, Store group of objects in collection object. The collection framework of Java is shown in Figure 17.5.



**Figure 17.5** Collection class of `java.util`

Collection class provides several interfaces. They are shown in Table 17.5.

**Table 17.5** Collection class interfaces

Collection Class Interface	Classes
Set<T>	HashSet<T> LinkedHashSet<T>
List<T>	Stack<T> LinkedList<T> ArrayList<T> Vector<T>
Queue<T>	LinkedList<T>
Map<k, s>	HashMap<k, s> Hashtabel<k, s>

In our next section, we deal with vector class and leave the balance to be dealt by other succeeding chapters on Java Collection.

### *17.5.1 Vector Class*



Vector is a data structure provided by java J2SE 5.0 to store any number of objects of any type, and these can be decided and resources can be allocated at run-time. The advantages of vectors over arrays can be summarized as follows:

- Vectors are dynamic and can be used to store any number, any type and any size of objects.
- Addition and deletion of objects from the vector is easy whereas deletion and addition in array requires readjustment of data.

The disadvantage is that only objects can be stored and not direct data. Hence we need to use wrapper class to convert primitive data to objects. We will implement vector class as it is logically linked to arrays and leave the balance topics for handling in ensuing chapters.

### *17.5.2 Vector Methods*

Methods supported by Vector Class are placed in Table 17.6.

**Table 17.6** Vector methods

Method	Remark
--------	--------

boolean addElement()  boolean assElementAt(n)	Adds at end  Adds at position n
boolean removeElement()  boolean removeElementAt(n)  boolean removeAllElements()	Removes the specified element  Removes element at position n  Removes all elements
int size()  element get(pos)  void set(pos,obj)	Returns no of object  Returns element at pos  Sets obj at position
Object[] toarray()	Copies contents into array

InsertElementAt ( )	Returns object class type array with all elements of vector
---------------------	---

## Example 17.13: VectorDemo.java: A Program to Demonstrate Usage of Vector Class

```

1. package com.oops.chap17;
2. import java.util.*;
3. public class VectorDemo {
4. public static void main(String[]
args) {
5. // create an object of vector
class to store integers
6. Vector<Integer> vec = new
Vector<Integer>();
7. // define an integer array
8. int [] x = new int []{
10,20,30,40,50,60,70};
9. /* store the integers into vec.
But integers are converted to objects
automatically by a feature autoboxing*/
10. for ( int i=0;i<x.length; i++)
11. vec.add(x[i]);

```

```

12.
System.out.println("\nDisplaying Vector
using vec.get()");
13.    for ( int i=0;i<vec.size();
i++)
14.        System.out.print(vec.get(i)+"
");
15.    System.out.println("Displaying
using ListIterator");
16.    ListIterator itr =
vec.listIterator();
17.    System.out.println("Using
ListIterator move forward");
18.    while (
itr.hasNext())System.out.print(itr.next(
)+" ");
19.    System.out.println("Using
ListIterator move reverse");
20.    while ( itr.hasPrevious())
21.
System.out.print(itr.previous()+" ");
22.    }// end of main
23.    }// end of vectorDemo

```

**Output:** Displaying Vector elements  
using vec.get()

```

10 20 30 40 50 60 70 Displaying
Vector elements using ListIterator
Using ListIterator move forward
10 20 30 40 50 60 70
Using ListIterator move reverse
70 60 50 40 30 20 10

```

---

<b>Line No. 6:</b>	<code>Vector&lt;Integer&gt; vec = new Vector&lt;Integer&gt; ();</code> creates an object called vec of <code>Vector&lt;Integer&gt;</code>
<b>Line No. 8:</b>	defines an array called x and assigns values to the array
<b>Line No. 9 to 11:</b>	adds elements into Vector object. Integers are converted to objects automatically by a feature autoboxing. Line No. 11 : <code>vec.add(x[i]);</code> shows addition to Vector object vec.
<b>Line No. 14:</b>	gets the elements from Vector object vec and prints them <code>System.out.print(vec.get(i) + " ");</code>
<b>Line No. 16:</b>	creates an object called <code>listIterator</code> for class <code>ListIteraotr</code> . This iterator is used to raverse the Vector both forward and reverse directions by using <code>itr.next</code> and <code>itr.previous</code> .

<b>Line Nos. 18 &amp; 20 :</b>	<code>checks if itr.hasNext or itr.hasPrevious is true</code>
--------------------------------	---

## 17.6 Summary

1. StringTokenizer class is used to receive multiple inputs in a single line.
2. Scanner class of Java is elegant and easy to use to read the input data from keyboard.
3. String.format() is a static method and hence can be used directly to control the output.
4. An array is defined as a collection of elements where each element is of the same type. The elements of the array can either be elementary data types like int, float, char, or can be complex data types like structures and objects.
5. Java.util package provides a useful collection class called **Arrays** class. Arrays class defines several static methods that can be used directly without creating an object.
6. A String is an array of characters or sequence of characters. A String is an object in Java. They are implemented by **String** class and **StringBuffer** class.

7. StringBuffer class is meant to fulfill the gap in requirements of programmers. We can insert in the middle or at the end of a string. Hence StringBuffer class elements are mutable.
8. StringBuilder class is the same as that of StringBuffer but it is not synchronized.
9. Collection framework is implemented in java.util package.
10. Vector is a data structure provided by java J2SE 5.0 to store any number of objects of any type.
11. Wrapper class converts the primitive data into objects.

## Exercise Questions

### Objective Questions

1. Arrays are passé to methods as pass by reference only. TRUE/FALSE
2. Which of the following statements are true in respect of Java Strings?
  1. Java String is an array of characters terminated by '\0'
  2. Java String is an object of String class.
  3. String is implemented using character array.
  4. Java String is implemented using String class or StringBuffer class.
3. A string in Java can be implemented by
  1. String class
  2. StringBuffer
  3. StringBuilder
  4. a, b and c

4. Which of the following are true in respect of Arrays and Strings?

1. arr.length() gives length of the array
2. stg.length; gives the length of the string
3. stg.length() gives the length of stg
4. arr.length ; gives the length of the array

1. ii and iv
2. i and iii
3. iii and iv
4. i and ii

5. String is a fixed length and content can be modified. TRUE/FALSE
6. StringBuffer class elements are immutable. TRUE/FALSE
7. StringBuilder class elements are mutable. TRUE/FALSE
8. Java String elements are mutable. TRUE/FALSE
9. Both StringBuffer and StringBuilder methods are synchronized. TRUE/FALSE
10. Vectors cannot store direct primitive data. TRUE/FALSE

#### **Short-answer Questions**

11. What is an array? How can it be created and set initial values?
12. Distinguish vector and array.
13. What is a Scanner class?
14. Distinguish the ways in which we can get length of an array and String.
15. How is an element of character array accessed?
16. How is an element of String accessed?
17. Explain formatted output using System.out.printf() method.
18. Explain String.format() method to output formatted data.
19. What is collection framework provided by Java?



20. What is wrapper class?

#### **Long-answer Questions**

21. Explain with examples how primitive data can be inputted from KeyBoard.
22. Explain the utility of StringTokenizer class. How is Scanner class different from class?
23. Write in detail about one-dimensional and multi-dimensional arrays. Also write about how initial values can be specified for each type of array.
  1. In what way is array different from ordinary variable?
  2. What conditions must be satisfied by the entire elements of any given array?
  3. What are subscripts? How are they written? What restrictions apply to the values that can be assigned to subscripts?
  4. What advantage is there in defining an array size in terms of a symbolic constant rather than a fixed integer quantity?
24. Explain the various methods available for creation and setting initial values to an array with suitable examples.
25. How are multi-dimensional arrays defined? Compare with the manner in which one-dimensional arrays are defined.
26. Explain salient features of Vector class.
27. Explain with examples the various methods to create and set initial values to a String.
28. Explain methods of String class with examples.
29. Explain what you understand by the statement StringBuffer elements are mutable.
30. Distinguish the String class and StringBuffer Class and StringBuilder Class.

#### **Assignment Questions**

31. Write a Java program to find the sum of elements of an array with recursion.
32. Write a Java program to find the product of two matrices.

33. Write a Java program to find the determinant of a matrix.
34. Write a function to merge two sorted arrays.
35. Write a program to read a line from the keyboard and print the line using a suitable encryption. Simple encryption can be a substitution with the next character A with B, a with b, z with a and so on. De-crypt and display the original message.
36. Develop String class to implement all the functionality provided by Java String class.
37. Develop Array class to implement the functionality provided by Arrays class.

### **Solutions to Objective Questions**

1. True
2. c
3. d
4. c
5. False
6. False
7. True
8. False
9. False
10. True

# 18

## Class Objects and Methods

### LEARNING OBJECTIVES

*At the end of this chapter, you will be able to understand and use*

- Java's classes and objects.
- Constructors and their overloading.
- Java's methods and their usage.
- This operator.
- Inner classes.
- Finalize method and Garbage collector.
- Static methods and classes.

### 18.1 Introduction

Java is a pure object-oriented language and hence all programming is enclosed in classes. Object-oriented programming involves writing programs that use classes. We create objects of different classes to solve the problem at hand and make objects communicate with each other through the member functions.

In this chapter, we will learn how to solve problems by using classes and objects. We will also cover topics such as data hiding, abstraction, access privileges enjoyed by members of the class, and concepts of constructor and destructor. Variations in methods such as methods definition, recursion and factory methods are also discussed. Classes within a class, called inner classes, are introduced. *This* operator and static member functions and their relevance and usage are explained. Java's garbage collector and finalizer methods are also discussed.

## 18.2 Classes and Objects

Look around your classroom. There are students all round. First of all, why have they been grouped together by your principal? Firstly because they all share **common interests**, for example they would like to learn the language Java or they would like to complete their PG or UG studies. In computer parlance, we can say that all students have the same functionality. That is why they could be grouped together.

But notice that each student has his own individual attributes. Attributes mean own member data like height, weight, marks, attendance, etc. Also notice that there are about 60 students in your class, each with his or her own attributes but common functionalities. We can call 60 instances of **object of Students class**. Well so much background for analogy. Figure 18.1 shows attributes and methods of class student.

Student
<pre>//Methods public void GetData() public void ComputeGrade() public void DisplayData()</pre>
<pre>//data members private String name; private int rollNo; private float totalMarks; private String grade;</pre>

**Figure 18.1** Class methods and attributes

**Class:** A collection of objects. We can also define as an array of instances objects. But class can have member functions and member data. Here, unlike array, a class can have different data types as its elements.

**Object:** An object is an entity. That is, it can be felt and seen. Examples are students, pens, tables, chairs, etc. Therefore, an object is an independent entity that has its own member data and member functions.

While methods of a class are common to all instances of the object, each instance of an object has its own set of attributes.

Hence, member data is also called instance variables, i.e. they are different for each instance of an object.

**Data Hiding/Data Abstraction.** It is customary to declare all member data as private only. But the data declared as private is hidden and cannot be accessed by anyone. This feature is called data hiding or data abstraction. But how can we get them? You can access this only through public member methods.

**Encapsulation.** An object-oriented programming like Java is a data primacy language i.e. data is important and functions are not important. As per memory mapping, data is stored in data area, such as stack and free space, and functions are stored in code area and there is a need to maintain strict control over the accessing of data by functions. Java achieves this control by using the encapsulation feature.  
*Encapsulation is binding member data and calling function together with security classification, so that no unauthorized*

*access to data takes place.* Java depends heavily on access specifiers to maintain data integrity. The security access specifiers are ***public, private, protected and default specifier package.***

### 18.3 Declaring a Class and Creating of Instances of Class Variables

Use keyword **class** followed by brace brackets. In side brace brackets we can include member data and member methods, as shown below. We will create a class for Student called `Student3`. The attributes are `name`, `rollNo`, `total marks` and `grade`. The grades are awarded based on total marks: A, if total marks are  $>80$ ; B, if total marks are  $>60$ ; C, if total marks are  $<60$ ; and D, if total marks are  $<40$ .

#### **Example 18.1: How to Declare a Class Called Student**



---

```
1. package com.oops.chap19;
2. import java.io.*;
3.   class Student1{
4.     // member data or attributes
5.     String name=""; // name of the
student
6.     int rollNo;
7.     double totalMarks;
8.     String grade="";
9.   }
10.  class StudentDemo {
11.    public static void main(String[]
args) {
12.      // create an object of class
Student3
13.      Student1 std = new Student1();
14.      std.name="Gautam";
15.      std.rollNo=6274;
16.      std.totalMarks=98.78;
17.      //compute grade
18.      if( std.totalMarks >=80.0)
std.grade="A";
19.      else if (std.totalMarks>= 60.0)
std.grade="B";
20.      else if (std.totalMarks>= 50.0)
std.grade="C";
21.      else std.grade="D";
22.      System.out.println("Students
Details are.....");
23.      System.out.println("Roll
Number:" + std.rollNo);
```

```

24.  System.out.println("Name:" +
std.name);
25.  System.out.println("Total
Marks:" + std.totalMarks);
26.  System.out.println("Grade:" +
std.grade);
27.  }
28.  }// end of StudentDemo
Output: Students Details are.....
Roll Number:6274 Name:Gautam Total
Marks:98.78 Grade:A

```

---

**L  
i  
n  
e  
N  
o  
s.  
5  
t  
o  
8  
:**

declare attributes like roll number, name, totalMarks and grade. As we have not declared access specifiers, these are considered public inside the package. This means that all the classes within the package can access these attributes.

**L  
i  
n**

declares a class StudentDemo with public static void() at Line No. 11. This means that is our main class and our program is thus named

**e  
N  
o.  
1  
o  
:**

`StudentDemo.java`. This is also called a driver program for class `Student2`

**L  
i  
n  
e  
N  
o.  
1  
3  
:**

`Student1 std = new Student1();` creates an object of class `Student1` named `std`. `New` statement allocates memory for `Student` object and then calls constructor (default constructor) if user does not explicitly define it.

**L  
i  
n  
e  
N  
o.  
s.  
1  
4  
-  
1  
6  
:**

show that we can assign data to member attributes with operator. For example `std.name="Gautam"`. Notice that we could directly access the member data because they are public to package.

## 18.4 Constructors

### *18.4.1 Default Constructor*

When you need to construct a house, you go to a specialist called architect or mason so that they make the house ready for occupation and usage. So is the case with class. When a class is declared and an object is created, the constructor for the class is called. Its job is to create the object, allocate memory for the data members and initialize them with initial values if supplied by the user. If no initial values are supplied by the user, then it initializes with default values. Similarly, if the user does not declare a constructor, default constructor is always created. Constructor will have the same name as that of the class but no return value. Constructor is always treated as public by default. Whenever an object is created, the constructor is automatically called.

---

```
Student Std = new Student();  
// default constructor is called.
```

---

## 18.4.2 Parameterized Constructor and Overloading of Constructors

Constructors are called parameterized when we pass initial values at the time of creation of the object: `Student Gautam = new Student (" Gautam", 6274, 74.00, "A") ;` we will demonstrate the concepts of constructors through an example of Polar class in Example 18.2:

### **Example 18.2: PassObjConst.java** **A Program for Showing Overloaded Methods**

```
1. package com.oops.chap19;
2. import java.io.*;
3. //import java.math.*;
4. class Polar2{
5. // all member data is declared as
private
6. private double r; // magnitude
7. private double t; // angle theeta
8. private double a; // real
```

```

9. private double b; // imaginary
10. public void Setr( double rd){
r=rd;}
11. public void Sett( double td){
t=td;}
12. public void Seta( double ad){
a=ad;}
13. public void Setb( double bd){
b=bd;}
14. //constructors
15. //Default Constructor
16. Polar2() {
17. System.out.println("Polar2
Default constructor...");
18.
Setr(0.0);Sett(0.0);Seta(0.0);Setb(0.0);
}
19. //parameterized constructor
20. Polar2(double ad, double bd){
21.
System.out.println("parameterized(a & b)
constructor.");
22.
Setr(0.0);Sett(0.0);Seta(ad);Setb(bd);}
23. Polar2(double ad, double bd ,
double rd, double td){
24.
System.out.println("parameterized(
a,b,r,t) constructor.")
25.
Setr(rd);Sett(td);Seta(ad);Setb(bd);}
26. // public accessory functions

```

```

27.    public void ConvertToPolar()
{double x=0.0;
    28.    r=(a*a + b*b)*0.5;
x=Math.atan(b/a); //x in radians
    29.    t=Math.toDegrees(x);
    30.    } // to convert to polar form
    31.    public void DisplayPolar(){
    32.    System.out.println("Inside
DisplayPolar .");
    33.    /*directly use r and t because
accessed inside method */
    34.
System.out.println(«magnitude<r>:» +r +
«Angle<t>» + t);}
    35.    }// end of class Polar1
    36.    class Polar2Demo {
    37.    public static void
main(String[] args) {
    38.    // create an object of class
Polar1-Default constructor
    39.    Polar2 pol2 = new Polar2();
    40.    pol2.DisplayPolar();
    41.    Polar2 pol2a = new
Polar2(3.0,4.0); // 3 + J4
    42.    pol2a.ConvertToPolar();
    43.    pol2a.DisplayPolar();
    44.    Polar2 pol2b = new
Polar2(3.0,4.0,12.5,53.13); //3+J4 & r,  $\Phi$ 
    45.    pol2b.DisplayPolar();
    }
} // end of Polar2Demo

```

**Output** Inside Polar2 Default

```

constructor.....
    Inside DisplayPolar .
    magnitude <r> :0.0Angle<t>0.0
    Inside Polar2 parameterized(a & b)
constructor.....
    Inside Polar2 parameterized( a,b,r,t)
constructor.....
    Inside DisplayPolar.
    magnitude <r> :12.5Angle<t>53.13
    Inside DisplayPolar .
    magnitude <r>
:12.5Angle<t>53.13010235415598

```

---

<b>L i n e N o . 1 6 :</b>	declares a default constructor with null arguments. However, it calls Set methods to set initial values of 0.0 for a, b, r, t at Line No. 18. Note that these Set methods are defined at line Nos. 10–13.
--	---



Overloading of constructors means the same name but different signatures (number and type of parameters). In the above example, we have overloaded the constructors as shown below:

<b>Line No . 16:</b>	<code>Polar() { .....} Nil</code> arguments. This type of constructor is called NIL arguments constructor and also as default constructor.
<b>Line No . 20:</b>	<code>Polar2(double ad, double bd)</code> <code>{ .....} 2 arguments</code>
<b>Line No . 23:</b>	<code>Polar2(double ad, double bd , double</code> <code>rd, double td) {...} 4 args</code>

We have overloaded the constructors. Each overloaded constructor has a different number of arguments. Overloading means the same name but different signatures (number and type of parameters). When a message is sent, signatures are compared and best-fit methods get loaded. This happens at run time. We have to pass arguments to constructors. Hence, these are called parameterized constructors.

## 18.5 Specifying Private Access Specifiers and Use of Public Methods

### *18.5.1 Private Access Specifiers*

In the Class we have declared in Example 18.1, the attributes had no access specifiers and hence they had default access specifiers, i.e. they are public to all the members of package and private to outside package. However, policy of any good OOPS language is to declare the attributes as private and provide public methods to access these attributes. The next example shows that we have declared all the attributes as private

and provided public methods like `GetData()`, `DisplayData()` and `ComputeData()`.

### *18.5.2 Methods*

Once the object to a class is created, we can access the member functions and public member data of a class using the dot (.) operator.

In Java, class member functions are called **Methods**. A method is nothing but a code written to achieve a result. The syntax is:

```
Return Type    Function name (Argument List) ; // note the semicolon
  ↙           ↙           ↙
int      FindMax      ( int a , int b , int c );
```

`public void main ()` calls the function `FindMax()` by supplying the arguments and receives the result through return type.

```
ans = FindMax ( a, b ) ; // a,b,
and ans are all integer data types
```

---

**Arguments:** Number and type of parameters are called arguments.

Parameters are also called formal parameters.

**Signature:** Name and arguments together are called the signature of the method. A method will have a return type which is outside the signature. The variables declared inside a method are called local variables and are never available anywhere outside the method. A method can be executed by sending a message to the object. Message comprises – a reference to the object, a dot ( . ) , method name and actual parameters (arguments). In the next example, we will show usage of access specifiers and also methods that return parameters. Whenever IO Stream commands are used, there is a likelihood of errors and exceptions occurring. Hence, try and catch block.

**Instance Methods:** Methods are defined inside a class and can be accessed by object or instances of object. Hence, these methods are also called instance methods.

**Access/Mutator Methods.** Normally the member data is declared private and public accessory methods are declared inside a class to access them. There are a set of these public methods which are called `Get ()` and `Set ()` methods whose job is just to set the private data and `read (Get)` the private data. The `Get ()` methods are called accessor methods and the `Set ()` methods are called mutator methods. These are extensively used in deciding the behaviour pattern of the objects.

**Example 18.3: How to Declare a Class Called Student with Data with Access Specifiers and Function Accepts a Value as Parameter and Returns a Value**

---

```
1. package com.oops.chap19;  
2. import java.io.*;  
3. class Student3{
```

```

4. // member data or attributes
5. private String name=""; // name of
the student
6. private int rollNo;
7. private double totalMarks;
8. private String grade="";
9. //public metods
10.   public void GetData() {
11.   try{
12.   BufferedReader input = new
BufferedReader (new
InputStreamReader(System.in));
13.   System.out.println("Enter
Students name :");
14.   name = input.readLine();
15.   System.out.println("Enter
Students roll number :");
16.   rollNo=
Integer.parseInt(input.readLine());
17.   System.out.println("Enter total
marks :");
18.   totalMarks=
Double.parseDouble(input.readLine());
19.   }catch ( Exception e){}
20.   }//endof GetData()
21.   public void DisplayData() {
22.   System.out.println("Students
Details are.....");
23.   System.out.println("Roll
Number:" + rollNo);
24.   System.out.println("Name:" +
name);

```

```

25.    System.out.println("Total
Marks:" + totalMarks);
26.    System.out.println("Grade:" +
ComputeGrade(totalMarks));
27.    }
28.    public String
ComputeGrade(double totalMarks) {
29.        String grade="";
30.        if( totalMarks >=80.0)
grade="A";
31.        else if (totalMarks>= 60.0)
grade="B";
32.        else if (totalMarks>= 50.0)
grade="C";
33.        else grade="D";
34.        return grade;
35.    }
36.    }// end of Student3
37.    class Student3Demo {
38.        public static void main(String[]
args) {
39.            // create an object of class
Student3
40.            Student3 Gautam = new
Student3();
41.            Gautam.GetData();
42.            Gautam.DisplayData();
43.        }
44.    }// end of Student3Demo
Output: Students Details are.....
Roll Number:50595
Name:Ramesh

```

Total Marks:98.0

Grade:A

<b>Line Nos. 3 to 5:</b>	define class Student3 . Line Nos. 5 to 8 declare attributes for Student3 that are declared as private.
<b>Line Nos. 10, 21 &amp; 28:</b>	declare public methods for accessing private data and perform the required functionality like GetData() , Display Data() which in turn calls ComputeGrade() .
<b>Li</b>	declares a public function <b>public</b> String



<b>Line No. 28:</b>	Compute Grade(double totalMarks) that takes an argument called totalMarks and computes and returns grade of String data type in Line No. 34.
<b>Line Nos. 10 to 20:</b>	define a public method as public void GetData () {.....}.
<b>Line Nos. 11 &amp; 19:</b>	use try and catch block since GetData () uses stream objects and there is a likelihood of error/exception taking place. Catch block catches Exception Object 'e' at Line No. 19.
<b>Line</b>	BufferedReader input = <b>new</b> BufferedReader ( <b>new</b>

<b>Line No. 12:</b>	<code>InputStreamReader(System.in));</code> We are creating an object of <code>BufferedReader</code> called <code>input</code> , that is an instance of <code>InputStreamReader</code> . Observe that <code>System.in</code> represents keyboard.
<b>Line Nos. 13 &amp; 14:</b>	read a string from keyboard. Whereas Line Nos. 15 and 16 read a primitive data integer from keyboard. We have used <code>Integer.parseInt()</code> to read data from keyboard. Similarly, other data type can also be inputted.
<b>Line No. 26:</b>	<code>DisplayData()</code> calls a method <code>ComputeGrade()</code> <code>System.out.println("Grade:" + ComputeGrade());</code>
<b>Line No. 37:</b>	creates a class called <code>Student3Demo</code> . This is our main class to act as driver for <code>Student3</code> . This class contains public static <code>void main(String[] args){.....}</code> . At Line 40, we have created an object of <code>Student3</code> with the

	statement : Student3 Gautam = <b>new</b> Student3 ();
<b>Li</b> <b>n</b> <b>e</b> <b>N</b> <b>os</b> <b>.</b> <b>41</b> <b>&amp;</b> <b>4:</b>	show how to invoke methods with the object created.

### *18.5.3 Math Class of Java*

In our next example, we will show all the concepts we have discussed so far through a class declaration called Polar. In this program, you will learn the concept of class and object and public accessory functions. The class we will consider is vector in Cartesian form denoted by real number **a** and an imaginary **component b**. In Polar coordinates, the same can be represented by vector whose magnitude is **r** given by the formula  $\mathbf{r} = \sqrt{\mathbf{a}^2 + \mathbf{b}^2}$  and the direction is given by  $\theta = \tan^{-1}(\mathbf{b}/\mathbf{a})$ . Our program will

accept the real value **a** and imaginary component **b** through a public accessory function called `GetData()` and convert it to Polar form with magnitude **r** and angle  **$\theta$**  through a public accessory function called `ConvertToPolar()` and display the result through a function called `DisplayPolar()`. A word or two about the way we will declare and define functions. In the next example, we will show how we can use math function. Java defines a `Math` class. Some of the important and widely used methods are shown in Table 18.1.

**Table 18.1** Math class of Java – some widely used methods

Method	Description
--------	-------------

static double sin(double arg)  //cos & tan also available	Returns sine cosine and tan of argument in radian
static double atan(double arg)  //asin() and acos available	Returns angle whose tan() is argument
static double sinh(double arg)  //cosh & tanh also available	Returns hyperbolic sine of angle given by argument.
static double pow (double y, double x)	Returns $y^x$
static double sqrt(double x)	Returns square root of x
static double exp(double x)	Returns $e^x$
static double log(double x)	Returns natural log of x
static double	Returns log base 10 of x

<code>log10(double x)</code>	
<code>static double abs(double x)</code>	
<code>static double ceil(double x)</code>	Returns smallest whole number $\geq x$
<code>static double floor(double x)</code>	Returns largest whole number $\leq x$
<code>static double rint(double x)</code>	Returns integer nearest to $x$ .

## Example 18.4: Polar1Demo.java A Program to Accept Data for Cartesian Vector $a + j b$ and Display the Result in Polar Form $r$ Angle $\theta$

```

1. package com.oops.chap19;
2. import java.io.*;
3. class Polar1{
4. // all member data is declared as
private
5. private double r; // magnitude

```

```

6. private double t; // angle theeta
7. private double a; // real
8. private double b; // imaginary
9. // public accessory functions
10. public void GetData(){
11.     try{
12.         BufferedReader input = new
BufferedReader (new
InputStreamReader(System.in));
13.         System.out.println("Enter
Cartesian <real number a>:");
14.         a =
Double.parseDouble(input.readLine());
15.         System.out.println("Enter
Imaginary <b> :");
16.         b=
Double.parseDouble(input.readLine());
17.         System.out.println("Enter
magnitude <r> :");
18.         r=t=0.0; // in side a member
method.r & t can be accessed d
19.     }catch( Exception e){}
20.     }
21. public void ConvertToPolar(){
22.     double x=0.0;
23.     r=Math.sqrt(a*a + b*b);
24.     x=Math.atan(b/a); // x is in
radians, b/a in radians
25.     t= (7.0/22.0)*180.0*x; // PI
radians equals 180 degrees
26.     t=Math.toDegrees(x); //conver
to degrees

```

```

27.    } // to convert to polar form
28.    public void DisplayPolar(){
29.        System.out.println("Inside the
classes member function .");
30.        /*We can directly use r and t
because it is inside the
31.        member function of the class
thus equal to public*/
32.
System.out.println(«magnitude<r>:» + r +
«Angle<t>» + t);
33.    } // to display in polar and
cartesian forms
34.    }// end of class Polar1
35.    class Polar1Demo {
36.        public static void
main(String[] args) {
37.            // create an object of class
Polar1
38.            Polar1 pol1 = new Polar1();
39.            pol1.GetData();
40.            pol1.ConvertToPolar();
41.            pol1.DisplayPolar();
42.        }
43.    }// end of Polar1Demo

```

**Output:** Enter Cartesian form Vector  
Details <real number a>: 3  
Enter Imaginary <b> : 4  
Enter magnitude <r> :  
Inside the classes member function .  
magnitude <r>  
:12.5Angle<t>53.13010235415598



<b>Line Nos. 5 to 8:</b>	declare the attributes as private and type double.
<b>Line Nos. 10 to 20:</b>	defines a method for <code>GetData()</code> that obtains data from keyboard using <code>Biffered Reader</code> at Line No. 12. It also uses try and catch blocks at Line Nos. 11 and 19.
<b>Line Nos. 23, 24 and 26:</b>	use <code>Math</code> class methods like <code>sqrt</code> , <code>atan</code> and <code>toDegrees</code>

Notice that when we refer to private data `a,b,r,t` inside a method like `ConverPolar()` and the method is invoked by the object, then these private variables can be accessed directly like we have done at

Line No. 31 and we do not need public accessory methods.

---

```
System.out.println("magnitude<r>:" + r +  
"Angle<t>" + t);
```

---

## *18.5.4 Call by Value and Call by Reference*

### **18.5.4.1 Call by Value**

Mode of data transfer between calling method and called method is carried out by copying variables listed as arguments into called method stack area, and subsequently, returning the value by called function to calling function by copying the result into stack area of the main method. This is called ***call by value***. Note that as copying of the variables, both for forward transfer and return transactions are involved, it is efficient only if values to be transferred are small in number and of basic data type. Accordingly, Java uses call by value for transferring all primitive data types.

As only copies are forwarded to method, the changes made to these variables by

called method are local changes and are not reflected back to calling method.

## Example

### 18.5: PassByValDemo.java – Passing Arguments by Call by Value

```
//PassByValDemo.java
1. package com.oops.chap19;
2. import java.io.*;
3. class CallByVal{
4. public void SwapData(int x, int y)
{
    5. System.out.println("Inside
SwapDataBefore Swap.....");
    6. System.out.println("x:" + x +
"y:" + y);
    7. int temp=x;x=y;y=temp;
    8. System.out.println("Inside
SwapData, After Swap");
    9. System.out.println("x:" + x +
"y:" + y); }
10. }// end of class callBy Val
11. class PassByValDemo {
12. public static void main(String[]
args) {
```

```

13.  int x=10;int y=100;
14.  CallByVal val = new CallByVal();
15.  val.SwapData(x,y);
16.  System.out.println("Inside main
:return from SwapData, ");
17.  System.out.println("x:" + x +
"y:"+y);}
18.  }

```

**Output:** Inside SwapDataBefore Swap.....

x:10y:100

Inside SwapData, After Swap .....

x:100y:10

Inside main on return from SwapData,

x:10y:100

<b>Line No. 15:</b>	we have passed data by call by value. Line Nos. 8 and 9 show that values inside Swapdata have indeed been swapped. These are local changes.
<b>Line Nos. 16 and 17:</b>	show that values of x and y remain unchanged after return from SwapData . This is characteristic of Call by Value.

#### 18.5.4.2 Call by Reference

For large data items occupying large memory space like objects, overheads like memory and access times, etc., become very high. Hence, we do not pass values as in call by value; instead, we pass the address to the function. Note that once the method receives a data item by reference, it acts on data item and the changes made to the data item also reflects on the calling function. This is like passing the address of the original document and not a photocopy. Therefore, changes made on the original document apply to the owner of the document as well.

Java passes objects only by reference. When an instance of object is created, we are creating the reference to object. As reference of the objects is passed, changes made to these objects by called program are reflected back to the calling program as well. Please make a note that reference itself is passed by call by value.

#### *18.5.5 Passing and Returning of Objects To and From Methods*

We can pass objects as arguments to a method and receive object from a method through return statement. In the example that follows, a method called StdGrade receives the object called Student as argument and computes the grade and returns the object to calling methods.

### **Example 18.6: StdGradeDemo.java** **A Program that Passes and Receives Objects To and From a Method**

```
//StdGradedemo.java
1. package com.oops.chap19;
2. import java.io.*;
3. class StdGrade{
4. // member data or attributes
5. String name=""; // name of the
student
6. int rollNo;
7. double totalMarks;
8. String grade="";
9. public void DisplayData() {
10.     System.out.println("Students
```

```
Details are.....");
    11.    System.out.println("Roll
Number:" + rollNo);
    12.    System.out.println("Name:" +
name);
    13.    System.out.println("Total
Marks:" + totalMarks);
    14.    System.out.println("Grade:" +
grade);
    15.    }
    16.    public StdGrade
ComputeGrade(StdGrade std) {
    17.        String grade="";
    18.        if( totalMarks >=80.0)
std.grade="A";
    19.        else if (totalMarks>= 60.0)
grade="B";
    20.        else if (totalMarks>= 50.0)
grade="C";
    21.        else grade="D";
    22.        return std;
    23.    }
    24.    }
    25.    class StdGradeDemo {
    26.        public static void
main(String[] args) {
    27.            // create an object of class
Student3
    28.            StdGrade std1 = new StdGrade();
    29.            std1.name="Gautam";
    30.            std1.rollNo=6274;
    31.            std1.totalMarks=98.78;
```

```

32.    //compute grade by passing
object and receiving back object
33.    std1= std1.ComputeGrade(std1);
34.    std1.DisplayData();
35.    }
36. }// end of StudentDemo

```

**Output:** Students Details are.....

Roll Number:6274 Name:Gautam Total  
Marks:98.78 Grade:A

**L** shows the function StdGrade ( StdGrade  
**i** std) receiving the object and Line No. 22  
**n** returns the object. Please note that this transfer  
**e** of the object is by Pass by Reference and hence  
changes made inside a method are reflected  
**N** back to the calling program.  
**o**  
**.**  
**1**  
**6**  
**:**

As a second example of object passing, let us consider a case wherein we pass object to a constructor. We pass object to a constructor



when we need a clone(copy) of the object initially.

---

```
Student std1 = new Student (
    "Ramesh", 50595, 98.7, "A") //
    name, no, total, grade
Student std2 = new Student(std1); //
std2 is a clone of std1
```

---

## Example 18.7: PassObjConst.java A Program for Showing Overloaded Methods

```
//PassObjConst.java
1. package com.oops.chap19;
2. import java.io.*;
3. class Polar4{
4. private double a; // real
5. private double b; // imaginary
6. public void Setab(double ad,
double bd) {a=ad;b=bd;}
7. //parameterized constructor
8. Polar4(double a, double b)
{Setab(a,b);}
```

```

9. // public accessory functions
10. Polar4(Polar4 pol){ a=
pol.a;b=pol.b;}//pass object
11. public void DisplayPolar(){
12.   System.out.println("Inside
DisplayPolar .");
13.   System.out.println("Real <a> :"  
+ a + "Imag <b>" + b);
14. } // to display in cartesian  
form
15. }//end of class
16. class PassObjConst {
17. public static void main(String[]  
args) {
18.   Polar4 pol4 = new  
Polar4(3.0,4.0); // 3 + J4
19.   System.out.println("Given Vector  
in Cartesian form .");
20.   pol4.DisplayPolar();
21.   Polar4 pol4a = new Polar4(pol4);  
// we have passed pol4 object to make a  
clone of pol4
22.   System.out.println("Cloned  
Vector pol4a Cartesian form .");
23.   pol4a.DisplayPolar();
24. }
25. }// end of PassObjConst

```

**Output:** Polar4 parameterized(a & b)  
constructor.....

Given Vector in Cartesian form .

Inside DisplayPolar .

Real <a> :3.0Imag <b>4.0

Cloned Vector pol4 in Cartesian form

.

Inside DisplayPolar .

Real <a> :3.0Imag <b>4.0

---

<b>L</b>	Polar4(Polar4 pol){ a=
<b>i</b>	pol.a;b=pol.b;} and statement 21
<b>n</b>	Polar4 pol4a = <b>new</b> Polar4(pol4);
<b>e</b>	shows passing of the object to a constructor.
<b>N</b>	Pol4a ia clone of pol4
<b>o</b>	
<b>.</b>	
<b>1</b>	
<b>o</b>	
<b>:</b>	

### *18.5.6 Method Overloading*

The same name for the method but with different number of arguments or different type of arguments, we can say methods are overloaded. Overloaded methods can have different types of return types. When the signature of the method, i.e. name and

arguments, match, Java loads the particular version of the method into primary area. While there can be several versions of a method compiled, they are loaded at run time as per arguments supplied by the user, thus saving the primary memory. Consider the following examples of overloading because name is the same but there are different number of arguments.

---

```
public double Find(double side){ return
side*side;}//Area of square
public double Find(double l, double b ){
return l*b;}//rectangle area
public double Find( double l , double b
, double h){ return l*b*h;}
// cube vol
```

---

A second set of examples are for different types of arguments for overloaded methods

---

```
public int Find( int a , int b){ return
( a<b) ? a:b;}// smaller of a & b
integers
public double Find( double a , double b)
{ return ( a<b) ? a:b;}// smaller
of a & b double
```

---

Once you have declared the above overloaded methods, the following statements are not overloaded methods and lead to error. Why?

---

```
public float Find(double l, double b)
{return l*b;}//rectangle area
public short Find( int a, int b){ return
( a<b) ? a:b;}// smaller of
a & b integers
```

---

## Example 18.8: Find.Java To Show Overloaded Methods

```
//FindDemo.java
1.    package com.oops.chap19;
2.    import java.io.*;
3.    class OverLoadedFind{ // over
loaded Find for square, rect, cube
4.        public double Find( double
side){ return side*side;}
5.        public double Find( double l ,
double b ){ return l*b;}
6.        public double Find(double l,
```

```

double b, double h){return l*b*h;}
7.      }
8.      class FindDemo {
9.      public static void
main(String[] args) {
10.     OverLoadedFind ovrfind = new
OverLoadedFind();
11.     System.out.println(" Area of a
square: " + ovrfind.Find(10.0));
12.     System.out.println("Area of a
rect"+ ovrfind.Find(10.0,20.0));
13.     System.out.println(" Vol
ofCube:"+ovrfind.Find(10.0,20.0,30. 0));
14.     }
15.     }// end of FindDemo.java
Output: Area of a square: 100.0
Area of a rectangle: 200.0 Vol of a
Cube: 6000.0

```

<b>Line Nos. 4, 5 &amp; 6:</b>	define overloaded methods to determine area of square, rectangle and volume of the cube.
<b>Line Nos. 11, 12 &amp; 13:</b>	use overloaded Find. Note that Java automatically understands the data types as we have declared prototypes in statements 4 to 6.

### 18.5.7 Recursion

Recursion is an advanced feature supported by Java language. Recursion means a method calling itself. Consider the problem of finding a factorial of a number. In the example, we can clearly see that `Fact (5)` is calling `Fact (4)` and so on.

```
Fact (5) = 5 × 4 × 3 × 2 × 1
          = 5 × Fact (4)
          = 5 × 4 × Fact (3)
          = 5 × 4 × 3 × Fact (2)
          = 5 × 4 × 3 × 2 × Fact (1)
          = 5 × 4 × 3 × 2 × 1 × Fact (0)
          = 5 × 4 × 3 × 2 × 1 (Fact (0)
=1)
```

**Example 18.9: RecursionFact.Java  
To Calculate Factorial Using  
Recursion**

---

```
1. //RecursionFact.java
2. package com.oops.chap19;
3. class Factorial{
4. public long factrecur(int
a) //function definition*/
5. { long ans;
6. if(a<=0) return(1); // fact(0) = 1
by definition
7. else ans =a*factrecur(a-1);
//method call with recursion*/
8. return(ans); //returns the value
of fact to main*/
9. } //end of function fact*/
10. } // end of class Factorial
11. class RecursionFact{
12. public static void
main(String[] args) {
13.     Factorial fact = new
Factorial();
14.     System.out.println(" Fact of 5
: "+ fact.factrecur(5));
15.     System.out.println(" Fact of 7
: " + fact.factrecur(7));
16.     System.out.println(" Fact of 10
: " + fact.factrecur(10));
17. }
18. } //end of recursionFact
Output : Factorial of 5 : 120
Factorial of 7 : 5040
Factorial of 10 : 3628800
```

---



<b>L</b>	defines <code>factrecur(int a)</code> to find the
<b>i</b>	factorial by recursion. If you call the method
<b>n</b>	with argument 1, it will return 1 else it returns
<b>e</b>	<code>factrecur(a-1) * a</code> . This means that we now
<b>N</b>	call <code>factrecur</code> with <code>a-1</code> as argument, i.e. the
<b>o</b>	method calls itself.
<b>.</b>	
<b>4</b>	
<b>:</b>	

## 18.6 Usage of *this* Keyword

We are aware that memory management of Java allocates separate memory area called code section for your code. Accordingly, your method code will be in this area. But we also know that variables are stored in stack area. When a method is invoked by an object, there is a need to link up method that is called with the objects data. Therefore, the object is forwarded to method as an argument to that method. But this argument, i.e. address of the object invoking

the method, is not visible like ordinary formal arguments and it is hidden. This hidden address is called **this object**. It is called **this** because it points this object, that is, the object that has invoked the method. In a method if you use *this keyword* explicitly, then it refers to object member data. Note that we can get the same effect by using the object instead. But note that **this** has all the advantages of instant and fast access to object. Line No. 22 in the next example shows this feature.

A second use of keyword **this** is to resolve naming conflicts. For example, if our method had the following definition: **public void** Setr( double r) { r=r; }, there is naming conflict. Keyword **this** comes to the rescue. We could write the method as:

```
public void Setr(double r)
{ this.r=r; } Line Nos. 8 to 11 show this
feature. Look at the program in Example
18.5 rewritten using this operator.
```

## Example 18.10: PolarThis.java To Show the Uses of *this* Operator. Constructors and Destructors of a Class

```
1. package com.oops.chap19;
2. import java.io.*;
3. class Polar3{
4.     private double r; // magnitude
5.     private double t; // angle theeta
6.     private double a; // real
7.     private double b; // imaginary
8.     public void Setr(double r)
9.     {this.r=r;}
10.    public void Sett(double t)
11.    {this.t=t;}
12.    public void Seta(double a)
13.    {this.a=a;}
14.    public void Setb(double b)
15.    {this.b=b;}
16.    //constructors Default
17.    Constructor
18.    Polar3() {
19.        System.out.println("Polar3
20.        Default const..");
21.    }
```

```

Setr(0.0);Sett(0.0);Seta(0.0);Setb(0.0);
}
16.    Polar3(double a, double b){
17.
System.out.println("parameterized(a & b)
const...");
18.
Setr(0.0);Sett(0.0);Seta(a);Setb(b);}
19.    // public accessory functions
20.    public void ConvertToPolar()
{double x=0.0;
21.    r=(a*a + b*b)*0.5;
x=Math.atan(b/a); //x in radians
22.    t=Math.toDegrees(x);
23.    } // to convert to polar form
24.    public void DisplayPolar(){
25.    System.out.println("Inside
DisplayPolar .");
26.    System.out.println("mag<r> :"+
this.r +"Angle<t>"+ this.t);
27.    } // to display in polar and
cartesian forms
28.    } // end of class Polar1
29.    class PolarThis1Demo {
30.    public static void
main(String[] args) {
31.    Polar3 pol3a = new
Polar3(3.0,4.0); // 3 + j4
32.    pol3a.ConvertToPolar();
33.    pol3a.DisplayPolar();
34.    }
35.    } // end of Polar2Demo

```

**Output:** Inside Polar3 parameterized(a  
& b) constructor.....  
Inside DisplayPolar.  
magnitude <r> :12.5Angle<t>53.13

---

## 18.7 Garbage Collection

Java resorts to automatic memory release for objects no more needed. Java uses automatic garbage collector to manage memory. So what is automatic garbage collector?

Garbage collector is a software program that runs in the background and picks up and clears memory space occupied by the dead objects. It follows a predetermined algorithm to decide which objects have no remaining reference, i.e. they are no longer referred and frees the memory automatically. This process will happen when the program is idle or when there is insufficient free memory to be allotted.

## 18.8 Finalizer and Finalize() Methods

Java Run Time Environment calls `Finalize()` method just before the object is freed. An object may be holding several resources like handler to window and files, etc. We need to release these resources prior to the object's release. This job is done when Garbage collector handles the object and it automatically calls `finalize()` method.

---

```
protected finalize() { //write all the  
    code for release of software }
```

---

Make a note that the time of execution of `finalize()` is not known as it is executed whenever Garbage collector calls for it and Garbage collector follows random algorithm to release the object. Keyword `protected` to ensure code is not available outside the class.

## 18.9 Final Variable

Many a time, we need variables to hold a constant value and this does not change during the execution of the program. Then we need to use final specifier. For example,

consider **private final double** `PI` = `3.141517`; this means that we have declared a private variable `PI` which holds a double data type `3.141517`. We want this value to be constant. Hence, we use the qualifier `final`.

## 18.10 Access Control and Accessing Class Members

The security access specifiers are `public`, `private`, `protected`, and default specifier `package`. `Protected` specifier is applicable in case of inheritance relationship.

**Public:** Member functions and data if any declared as **public** can be accessed outside the class member functions. This means that public member can be accessed by any other code. For example, we have defined `main()` as public, as it is called by Java outside the program.

**package (default):** You can access this from any other class in the same directory. Note that when no access specifier is

specified, default is public up to package level, but private to members outside the package. We can refer to these default or public member data directly without the help of public function defined in the class.

**Private:** Member data declared as **private** can only be accessed within the class member methods and data is hidden from outside. We cannot access private members directly. Instead, we need public accessory methods to access these data.

**Protected:** Member data and member functions declared as **protected** is private to outsiders and public to descendants of the class in **inheritance** relationship. You will learn more about this in the inheritance chapter that follows.

### **Example 18.11: AccessControl.java. Constructors and Destructors of a Class**



---

```
//EmpAccess.java.  
1. package com.oops.chap19;  
2. class Employee{  
3. String name; // default access .  
Package level  
4. public int rollNo; // public to  
all  
5. public String dept; // public  
6. private double sal; //private  
7. public double GetSal(){ return  
sal;}  
8. public void SetSal(double s){ sal  
= s;}  
9. }// end of class  
10. class EmpAccess{  
11. public static void  
main(String[] args) {  
12. Employee emp = new Employee();  
13. emp.name="Ramesh"; // allowed  
since name is default  
14. emp.rollNo=50595; // allowed  
since rollNo is public  
15. //emp.sal=10000.00; // Not  
allowed since its private  
16. // use public function like  
SetSal() and GetSal()  
17. emp.SetSal(15000.00);  
18. System.out.println("Emp  
name:"+emp.name);//package default  
19. System.out.println("Emp id No:"  
+emp.rollNo);//public
```

```
20.    System.out.println("Emp Salary  
Set:"+emp.GetSal()); //public method  
21.    }
```

```
22.    } //end of RecursionFib
```

**Output:** Employess name: Ramesh

Employess id No: 50595

Employess Salary Set: 15000.0

<b>Line No. 3:</b>	String name. Default public to package members. Hence Line Nos. 13 & 18 are ok and allowed.
<b>Line Nos. 4 &amp; 5:</b>	declare public variables. Hence, Line Nos. 14 & 18 are ok.
<b>Line No. 6:</b>	declares sal as private. Hence, it is an error if you try to access like in Line No. 15 directly. Instead, you have to use public accessory method like we have done in Line Nos. 17 & 20.

---

## 18.11 Static Members

Many a time, you will need a member of a class that is independent of any object of the class. For example, you need to keep a count of number of instances that are created. Variable declared as static, though residing inside a class, can keep track of the objects being created. This means that it is accessible to all instances of the class. We can say that static data belongs to class and not to an object.

One important advantage of static declaration is that once a member is declared static, it can be called before the object is created. For example, when we have declared `void main()` as **public static void main( String[] args )**, Java Run Time Environment calls the `main()` function before creation of any object.

Static variables are global variables. Whenever an instance of object is created, no resource is allocated to static variables.

All instances of the class can have access to static variables. We can also declare a block as static block, in which case it can handle only static members.

Member methods can also be declared as static methods. But the rules of usage are as shown below:

- A static method can only call a static method
- Once a method is declared as static, all data being handled by the method also needs to be static
- Keywords super in case of inheritance and this operator cannot be used with static methods

### **Example 18.12: EmpStatic.java A Program to Demonstrate the Use of Static. Constructors and Destructors of a Class**

```
//EmpStatic.java
1. package com.oops.chap19;
2. class EmpStatic{
3. static int countEmp;
4. static int mcount;
5. static int fcount;
```

```

6.  static boolean mf;
7.  static double totalSal;
8.  // static block initilization
9.      static
10.     {System.out.println( "block
initializes all static:");
11.
countEmp=0;mcount=0;fcount=0;mf=true;tot
alSal=0.0; }
12.     static void Counts(boolean mf,
double s)
13.         {if (mf==true) ++mcount;
14.         else ++fcount;
15.         totalSal+=s;
16.         ++countEmp;}
17.     static void Display()
18.     {System.out.println("Total
Sal:"+EmpStatic.totalSal);
19.     System.out.println("Total emp
:"+ EmpStatic.countEmp);
20.     System.out.println("Male
employess :"+ EmpStatic.mcount);
21.     System.out.println("Female
employess :"+EmpStatic.fcount);}
22.     public static void
main(String[] args) {
23.         EmpStatic.Counts(true,
10000.00);
24.         EmpStatic.Counts(false,
12000.00);
25.         EmpStatic.Display();}
26.     }// end of EmpStatic class

```

**Output:** block initializes all static variable :

```
Total Sal:22000.0
Total No of employess:2
No of male employess:1
No of female employess :1
```

---

After EmpStatic is run, the first thing that happens is that all the static statements are executed. Line No. 10 Static block is executed and all static variables are initialized.

<b>L i n e N o . 2 3 :</b>	As a next step void main() is executed. Main() in turns calls Count() at line Nos. 24 and 25 and Display() at Line No. 26 which displays the count of employees and total salary dispersed.
--	---

A second important use of static declaration is that we can use static members outside the class in which they are declared. For this, you need to specify class name followed by dot. For example, `StaticOutClass.totalSal`; note that there is no need to use the object followed by dot, as we do for normal variables. Example 18.14 demonstrates this concept.

### **Example 18.13: EmpStatic2.java A Program to Demonstrate the Use of Static Outside the Class.**

#### **Constructors and Destructors of a Class**

```
//EmpStatic2.java
1.  package com.oops.chap19;
2.  class StaticOutClass{
3.    static int countEmp;
4.    static int mcount;
5.    static int fcount;
6.    static boolean mf;
7.    static double totalSal;
```

```

8.  // static block initilization
9.  static
10. {System.out.println( "block
initializes static vars :");
11.
countEmp=0;mcount=0;fcoun=0;mf=true;tot
alSal=0.0;}
12. static void Counts(boolean mf,
double s)
13. { if (mf==true) ++mcount;
14. else ++fcoun;
15. totalSal+=s;
16. ++countEmp;}
17. static void Display() {
18. System.out.println("total
Sal:"+StaticOutClass.totalSal);
19. System.out.println("Total empl:"+
StaticOutClass.countEmp);
20. System.out.println("male emp:"+
StaticOutClass.mcount);
21. System.out.println("female
emp:"+StaticOutClass.fcoun);}
22. }
23. class EmpStatic2{
24. public static void main(String[]
args) {
25. StaticOutClass.Counts(true,
10000.00);
26. StaticOutClass.Counts(false,
12000.00);
27. StaticOutClass.Display();
28. // enter more

```



```

29. StaticOutClass.Counts(true,
20000.00);
30. System.out.println( "Display
static vars outside class");
31. System.out.println("total
Sal:"+StaticOutClass.totalSal);
32. System.out.println("Total empl:"+
StaticOutClass.countEmp);
33. System.out.println("male emp:"+
StaticOutClass.mcount);
34. System.out.println("female
emp:"+StaticOutClass.fcount);
35. }
36. }// end of EmpStatic2 class

```

**Output:** block initializes all static variable :

```

total employess salary :22000.0
Total No of employess :2
No of male employess :1
No of female employess :1
Display static variable directly
outside class
total employess salary :42000.0
Total No of employess :3
No of male employess :2
No of female employess :1

```

<b>Line</b>	show use of static variables directly by
-------------	--

<b>Nos. 30 to 34:</b>	mentioning the class name, <code>StaticOutClass</code> , without using the object.
-------------------------------	--

## 18.12 Factory Methods

What does a factory do? It simply produces an output depending on the customers' choice. Factory methods in Java are designed to produce objects to the class to which factory method belongs. Factory methods are static methods. This means that they can be called directly without creating object.

The advantage of factory method is that it accepts arguments and returns the object of users' choice; thus factory method removes the need to create several constructors and overload them.

As an example we will consider `format()` method and `NumberFormat` class provided by `java.text` package. The steps involved are:

- Create an object `NumberFormat` class through usage of `getNumberInstance()` method

- Decide the format using  
setMaximumIntegerDigits()  
or setMinimumIntegerDigits() or  
setMaximumFractionDigits() or  
setMinimumFractionDigits()
- Fix the format using format() method

## Example 18.14: FactoryDemo.java To Show the Use of Factory Method and Numberformat Class. Constructors and Destructors of a Class

```
//FactoryDemo.java Factory Method usage
package com.oops.chap19;
import java.io.*;
import java.text.NumberFormat;
import java.util.*;
class FactoryDemo{
public static void main(String[] args)
{ final double PI = 3.1415197;
Scanner si = new Scanner(System.in);
System.out.print("Enter angle in
degrees:");
double degrees =si.nextDouble();
```

```
double degrad = (degrees * PI)/180.0;
System.out.println("Degrees in radians
with outFormatting:" + degrad);
NumberFormat numfrmt =
NumberFormat.getNumberInstance();
// format the number
numfrmt.setMaximumIntegerDigits(2);
numfrmt.setMaximumFractionDigits(2);
//convert to to string and display
String stg = numfrmt.format(degrad);
System.out.println("Degrees in radians
with Formatting using "+stg);
}
} //end of FactoryDemo
Output: Enter angle in degrees:180
Degrees in radians with out Number
Formatting:3.1415196999999995
Degrees in radians with Formatting using
Factory Method3.14
```

---

## 18.13 Nested Classes

Nested class means class within a class, also called container class in the literature.

Container class is one of the techniques provided by Java to achieve reusability of the code. Inheritance is another powerful feature for reusability which we will explore in the chapter on inheritance.

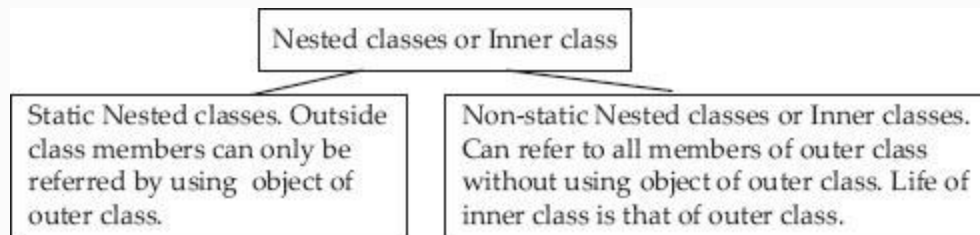
Container class means a class containing another class or more than one class. For example, a computer **has** a microprocessor in it. Further, a Student class can have a data member belonging to String class. Then we would say that String class is contained in Student class. In other words, it can also be called **composition or aggregation** of String class. The advantage of containment is that we can use String class within Student class to store the details of name and address, etc., belonging to String class. Similarly, if we define a class called Date with day, month and year information, we can define object of Date within the class called Student and define Date-related data such as DOB, DOJ, etc.

**Composition is a “has”-type of relation.** For the example we have given above, Student has name of string class and DOB of Date class. If class Date is defined inside a class called Student, we can call Date as inner class and Student as outer class. Rules of usage are as follows:

- Inner class is visible only in outer class and not outside the outer class.

- Nested class or inner class can access all members including private members of the outer class.
- Outer class does not have access to members of inner class.

The nested classes can be divided into static and non-static as shown in Figure 18.2.



**Figure 18.2** Nested classes

## 18.14 Inner Classes

### **Class within a Class – Inner Class:**

Inner class is one of the techniques provided by Java to achieve reusability of the code.

Inner class means a class containing another class or more than one class. Inner classes **cannot** be declared as static. There are four kinds of inner classes:

- **Member class:** This is defined just like a variable of a class. It has the same access specifiers as that of

variables.

- **Static member class:** Though it is defined inside a class with keyword `static`, it is actually outside the class. To refer to the static inner class from a class outside the containing class, use the syntax:

`outerClassName.InnerClassName`. A static member class may contain static fields and methods.

- **Local inner class:** A class defined within a method is called a local inner class. All rules regarding scope of variables of local variables also apply to this class. Local inner class cannot refer to non-local methods or variables.
- **Anonymous inner class:** A class created as parameter to a method that can either extend a class or implement an interface. The syntax is: `new Super(parameters) {methods}`

## Example 18.15: DateStudent.java Student Class Contains Date Class. Constructors and Destructors of a Class

```
//StdNested.java
1. package com.oops.chap19;
2. import java.io.*;
3. class StdNested{
4. // member data or attributes
```

```

5. String name="Ramesh"; // name of
the student
6. int rollNo=50595;
7. double totalMarks=98.5;
8. String grade="A";
9. void UseInner() {
10.     DateInner dateinner=new
DateInner();
11.     dateinner.DataDisplay();}
12.     //Now define inner class
DateInner
13.     class DateInner{
14.         int dd=15;int mm=10;int yy=10;
15.         void DataDisplay(){
16.             System.out.println("Students
Detail : outer class are.....");
17.             System.out.println("Roll
Number:" + rollNo);
18.             System.out.println("Name:" +
name);
19.             System.out.println("Total
Marks:" + totalMarks);
20.             System.out.println("Grade:" +
grade);
21.             System.out.println("DOJ from
inner class .....");
22.             System.out.println("DOJ
:"+dd+"/"+"mm+"/"+"yy);
23.         } //end of DataDisplay()
24.     }// End of class Dateinner
25. }//end of class StdNested
26. class StdNestedDemo {

```



```

27.    public static void
main(String[] args) {
    28.        StdNested stdouter = new
StdNested();
    29.        stdouter.UseInner(); }
    30.    }// end of StdNestedDemo
Output: Students Details from outer
class are.....
Roll Number:50595
Name:Ramesh
Total Marks:98.5
Grade:A
DOJ from inner class .....
DOJ :15/10/10

```

---

## 18.15 Summary

1. Java is a pure object-oriented language and hence all programming is enclosed in classes.
2. **Class:** A collection of objects or an array of instances objects.
3. **Object:** An object is an entity.
4. Methods of a class are common to all instances of the object; each instance of an object has its own set of attributes. Member data is also called instance variables.
5. Java's access specifiers are Public, Private, protected and default specifier packages.
6. When a class is declared and an object is created, the constructor for the class is called. Its job is to create the object, allocate memory for the data members and initialize them with initial values if supplied by the user.

7. Constructor will have the same name as that of the class but no return value.
8. Constructor is always treated as public by default. Whenever an object is created, the constructor is automatically called. Constructors are called parameterized when we pass initial values at the time of creation of the object.
9. There are two ways for forwarding data as arguments to methods. Call by Value and Call by reference. In Call by Value, parameters are copied on to called function area. Changes made to variables by called method are local and are not reflected into calling method. Java uses call by value for transferring all primitive data types.
10. In Call by Reference, addresses are passed to called methods. Changes made are reflected to calling method. Java uses Call by Reference to transfer objects as arguments to Methods.
11. Same name for the method but different number of arguments or different types of arguments, we can say methods are overloaded. Overloaded methods can have different types of return type.
12. Recursion is an advanced feature supported by Java language. Recursion means a method calling itself.
13. This operator refers to object that has invoked the method. It is also used to resolve naming conflicts.
14. Java resorts to automatic memory release for objects no more needed. Java uses automatic garbage collector to manage memory.
15. Java Run Time Environment calls `finalize()` method just before the object is freed.
16. Final keyword is specified to hold a constant value and does allow variable to change its value during the execution of the program.
17. Final keyword for a method is used when we want to prevent derived class overriding a base class method.

18. **Public:** Member functions and data if any declared as public can be accessed outside the class member functions. This means that public member can be accessed by any other code.
19. **Package (default):** You can access this from any other class in the same directory. Note that when no access specifier is specified, default is public up to package level, but private to members outside the package.
20. **Private:** Member data declared as private can only be accessed within the class member methods and data is hidden from outside.
21. We cannot access private members directly. Instead, we need public accessory methods to access these data.
22. **Protected:** Member data and member functions declared as protected is private to outsiders and public to descendants of the class in inheritance relationship.
23. Variable declared as static, though residing inside a class, can keep track of the objects being created. Static variables are global variables.
24. Advantage of static declaration is that once a member is declared static, it can be called before the object is created.
25. A static method can only call a static method. Once a method is declared as static, all data being handled by the method also needs to be static. Keywords super in case of inheritance and this operator cannot be used with static methods.
26. Nested class means class within a class, also called container class.
27. Non-static Nested classes or Inner classes. Can refer to all Members of outer class without using objects of the outer class. The life of inner class is that of the outer class.
28. Static Nested classes use keyword static. Hence, outside class members can only be referred to by using the object of outer class.

# Exercise Questions

## Objective Questions

1. Which of the following statements are true with respect to access specifiers of Java?

1. Default specifier in Java is private.
2. Default specifier is public.
3. Default specifier is public to package.
4. private outside the class, public inside the class.

2. Every class that is declared as public in Java is required to be stored in a file whose name is identical to the class name and with extension **.java** TRUE/FALSE

3. Which of the statements are true with respect to object creation using new keyword.

1. Object of class is created
2. Memory resources are allotted
3. Constructor is called
4. All of these

4. Which of the statements are true with respect to constructors?

1. Constructors have the same name
2. Constructors can return void
3. Constructors return no value
4. Constructors are private by default

1. i, ii and iv
2. i and ii
3. i and iii
4. i and iv

5. Which of the following statements are true with respect to recursive calls?

1. A method calls itself
2. Memory Overhead in the form of stack handling is present
3. Each recursive call creates own stack space
4. No overheads are present

1. i, ii and iii
2. i and ii
3. i and iv
4. i, ii and iv

6. Which of the following statements are true with respect to Garbage collector?

1. It runs in the background and releases objects no longer required
2. It is invoked by `finalize()` method
3. It runs on algorithm at fixed intervals and calls `finalize` method
4. Runs whenever memory available is insufficient

1. i, ii and iii
2. i, ii, iii and iv
3. i and iv
4. i, ii and iv

7. Which of the following statements are false?

1. Final variables are constant during run of the program
2. Final classes cannot be inherited
3. Final Methods can be inherited
4. Final variables can be inherited

1. i, ii and iii
2. i, ii, iii and iv
3. i and ii
4. i, ii and iv

8. Which of the following statements are false with respect to private and protected specifiers?

1. Protected variables can be inherited
2. Protected variables are public to package
3. Protected variables are private to non-inherited classes
4. Protected variables are public to own class

1. ii and iv
2. i, ii, iii and iv
3. i and ii
4. i, ii and iv

9. Which of the following statements are true with respect to static declaration?

1. Methods declared as static require objects to invoke them

2. Static Methods can be invoked directly using a class name and a dot operator.
3. Static variables reside in a class and belong to a class. Scope is local.
4. Static methods can only call static variables.

1. i, ii and iv
2. i, ii, iii and iv
3. ii and iv
4. i, ii and iv

10. Which of the following statements are true with respect to static declaration? Which of the following statements are true with respect to inner class declarations?

1. Inner class is public to outer class
2. Inner class can access all private members of outer class
3. Life of inner class life of outer class.
4. Anonymous inner class is a class created as parameter to a method

1. i, ii and iv
2. i, ii, iii and iv
3. ii, iii and iv
4. i and iv

#### Short-answer Questions

11. What are Java access specifiers? Explain default access specifier package.
12. What is default constructor? Is it the same as constructor with nil arguments?
13. Explain constructor overloading.
14. Which is better: call by value or call by reference?
15. Distinguish instance methods and constructors.
16. What is a mutator method?
17. Java does not support forwarding of addresses. Then how is call by reference achieved?
18. What are the uses of this operator in Java?
19. Discuss the functioning of Garbage collector.
20. What does `finalize()` method achieve?
21. What are static methods?
22. How does a static block differ from an ordinary block?

23. Explain the difference between class methods and instance methods.
24. What are the restrictions applicable to static methods?
25. What is an inner class?
26. Does out class have access to inner class variable? Why or why not?
27. What is an anonymous inner class?

#### **Long-answer Questions**

28. Explain how Java achieves encapsulation, data hiding and polymorphism. Explain with suitable examples.
29. Show how a class is defined in Java. Explain how instances of objects are created. What is the role of the new operator in the creation of instance of objects.
30. Explain method overloading with examples.
31. Explain Final keyword when used with variables, methods and classes with suitable examples.
32. Explain access control mechanism of Java.
33. Explain the working of nested classes in Java. What special restrictions apply to inner classes?
34. How are polymorphism and method overloading related? Discuss.
35. Explain the working of static methods, variables and block. What restrictions apply?

#### **Assignment Questions**

36. Write a Java program to accept the data of Student viz. name, roll no, branch through command line arguments and bifurcate the student as section A id his roll no<61 else as Section B. Use Swing components and Dialog boxes to display the result.
37. Write a program to check if the given year is a leap year or not. Display the result in dialog boxes. Use swing components.
38. Write a Java program to add two matrices.
39. Write a Java program to multiply two matrices.

## Solutions to Objective Questions

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



# 19

## Inheritance: Packages: Interfaces

### LEARNING OBJECTIVES

*At the end of this chapter, you will be able to understand and use concepts and programs relating to*

- Extending a class from an existing class, concepts in multilevel inheritance.
- Super- and subclass and overriding of superclass methods.
- Run-time polymorphism and abstract class.
- Packages and their usage.
- Interfaces.

### 19.1 Introduction

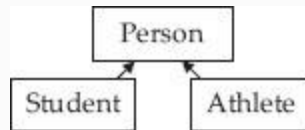
This chapter introduces you to one of the most powerful features of any objective-oriented language, i.e. inheritance.

Inheritance is a tool offered by Java to create another class from an existing class. We call this creation as extending a class. In the literature, the class that is a starting point or source of extension is called base class. Java calls this a superclass. The class that is extended is called derived class. In Java, derived class is called subclass. A subclass can inherit all the members, both methods and attributes, and also can implement its own methods. Further, a subclass can override superclass methods. We also introduce you to concepts of abstract class and interfaces which are part of Java's concepts for extending from existing classes.

## 19.2 Basic Concepts of Inheritance

Inheritance class hierarchy shown in Figure 19.1. Student and Athlete derive qualities from base class person. We say these subclasses have inherited from Person. Class Person is called base class and in Java it is

called Super, while Student and Athlete are called Subclasses. What can be inherited? Both member functions and member data can be inherited.



**Figure 19.1** Inheritance hierarchy

Inheritance can be incorporated into our program by using a keyword ***extends***. The following program shows a base class Super and a subclass Student.

**Example 19.1a: Write a Java Program to Create a Super Class Person for and a Subclass Called Student and Show Simple Inheritance Concept.**

---

```

1. package com.oops.chap19;
2. class Person {
3. // member data or attributes
4. String name; // name of the
student
5. int idNo;
6. //access and mutator methods
7. int GetIdNo(){ return idNo;}
8. void SetIdNo(int id){ idNo=id;}
9. String GetName(){ return name;}
10.    void SetName(String nm){
name=nm;}
11.    public void DisplayData() {
12.    System.out.println("Basic
Details are.....");
13.    System.out.println("Id Number:"
+ idNo);
14.    System.out.println("Name:" +
name);}
15.    }//end of class Person

```

---

<b>Line No. 1:</b>	defines our project package name as com.oops.chap19.
<b>Line Nos.</b>	define a base class Person. Line Nos. 4 and 5 declare two attributes of Person viz.

<b>2 to 15:</b>	<code>String name</code> and <code>int idNo</code> .
<b>Line Nos. 7 &amp; 9:</b>	are accessor methods and Line Nos. 8 and 10 are mutator methods.
<b>Line No. 11:</b>	defines a method called <code>DisplayData()</code> for displaying the basic data of person.

Compile the source file `Person.java` and store the class file `Person.class` in bin Directory. Refer to **Chapter 15** for comprehensive compilation and execution instructions both for Java and eclipse environments.

Now, another team member would like to reuse the code we have developed for class `Person`. He is aware of attributes and methods of class `Person`. In order to reuse he has to perform three tasks as shown in **Example 19.1b**:

- Indicate his own package details: **package** `com.oops.chap19;` Line No. 1

- Import the class mentioning the package: **import** `com.oops.chap19.Person`. Line No. 2
- Create a new class called Student from person by using the keyword **extends**: **class** Student **extends** Person as shown in Line No. 3:

## Example 19.1b: Student1Demo.java

### A program to show single inheritance from class Person in Example 19.1.

```
//Student1Demo.java
1.  package com.oops.chap19;
2.  import com.oops.chap19.Person;
3.  class Student1 extends Person {
4.  // member data or attributes
5.  private double totalMarks;
6.  private String grade;
7.  //access and mutator methods
8.  double GetMarks(){ return
totalMarks;}
9.  void SetMarks(double mks ){
totalMarks=mks;}
10.  String GetGrade(){ return
grade;}
```

```

11.    void SetGrade(String gd) {
grade=gd;}
12.    //public method
13.    public Student1
ComputeGrade(Student1 std) {
14.    if( totalMarks >=80.0)
std.grade="A";
15.    else if (totalMarks>= 60.0)
grade="B";
16.    else if (totalMarks>= 50.0)
grade="C";
17.    else grade="D";
18.    return std;
19.    }
20.    }//end of class Student
21.    // Driver class
22.    class Student1Demo{
23.    public static void
main(String[] args){
24.    // create an object of Student
25.    Student1 std = new Student1();
26.    std.SetMarks(97.8);
27.    std.SetName("Ramesh"); // we
have called super method
28.    std.SetIdNo(50595);
29.    std=std.ComputeGrade(std);
30.    std.DisplayData();
31.    System.out.println("Details of
Subclass Student: ");
32.    System.out.println("Total
Marks:" + std.GetMarks());
33.    System.out.println("Grade:" +

```

```
std.GetGrade() );}
```

```
34.    }//end of class Student1Demo
```

**Output:** Basic Details are.....

Id Number:50595

Name:Ramesh

Details from Sub class Student:

Total Marks:97.8

Grade:A

<b>L i n e N o s . 5 &amp; 6 :</b>	<p>define private member data. Note that we do not expect further inheritance from Student1. If inheritance further to next level is planned, we should declare them as protected. Accordingly, we have used accessor methods in Line Nos 32 and 33 to display these private variables.</p>
--	---

<b>L i n e N o</b>	<p>declared public method ComputeGrade(). Line No 30 classes it with std object. Also this method takes std1 object as argument and returns std1 object.</p>
--	--



• <b>1</b> <b>3</b> :	
<b>L</b> <b>i</b> <b>n</b> <b>e</b> <b>N</b> <b>o</b> • <b>3</b> <b>o</b> :	calls method belonging to superclass Person :std.DisplayData(). This is the real power of Inheritance. Student1 has not written code for DisplayData() but it is using the code of super.

### 19.3 Member Access Rules

Access specifiers play an important role in inheritance relationships. We have learnt that if we foresee inheritance and we need to allow subclass to access the super's members, we need to declare them as protected.

If we declare the members as private, then they are visible inside a class only. They are not visible to subclasses or outside the class.

Default access specifiers means they are public to package. This means that all the classes in the package can access them as if they are public.

Public specifier means they are accessible to all outside members.

### **Example 19.2a: Person2 . java With Protected and Private Data Members.**

```
1. package com.oops.chap19;
2. class Person2 {
3.     // member data or attributes
4.     protected String name; //protected
5.     protected int idNo;
6.     private double salary;
7.     //access and mutator methods
8.     int GetIdNo(){ return idNo;}
9.     void SetIdNo(int id){ idNo=id;}
10.    String GetName(){ return name;}
11.    void SetName(String nm){
name=nm;}
12.    double GetSal(){ return
salary;}
```

```

13.    void SetSal(double sl){
salary=sl;}
14.    public void DisplayData() {
15.        System.out.println("Person
Details are.....");
16.        System.out.println("Id Number:"
+ idNo);
17.        System.out.println("Name:" +
name);
18.    }
19.    public void DisplaySal(){
20.        System.out.println("Salary:" +
salary);}
21.    }//end of class Person

```

---

## Compile it as Person2 . java

<b>Li ne No s. 4 &amp; 5:</b>	we have declared as protected because we expect derived class to have access to the protected variables. Note that protected members are public to derived class and private to others.
<b>Li ne No s.</b>	we find no difficulty in directly accessing protected members because they are public to own class members.

<b>15</b> <b>–</b> <b>17:</b>	
<b>Li</b> <b>ne</b> <b>No</b> <b>s</b> <b>19</b> <b>&amp;</b> <b>21:</b>	we had to refer to private member salary only through a public method called <code>DisplaySal()</code> .

In the example that follows, we will show you `Student2Demo` class inheriting protected members and defining its own private data members. Private data members are not visible outside the class.

### **Example 19.2b: Student2Demo.java Inheriting From Super Person2**

```
//Student2Demo.java
1. package com.oops.chap19;
```

```

2.  import com.oops.chap19.Person2;
3.  class Student2 extends Person2 {
4.  // member data or attributes
5.  private double totalMarks;
6.  private String grade;
7.  //access and mutator methods
8.  double GetMarks(){ return
totalMarks;}
9.  void SetMarks(double mks ){
totalMarks=mks;}
10. String GetGrade(){ return grade;}
11. void SetGrade(String gd){
grade=gd;}
12.    //public method
13.    public Student2
ComputeGrade(Student2 std) {
14.    if( totalMarks >=80.0)
std.grade="A";
15.    else if (totalMarks>= 60.0)
grade="B";
16.    else if (totalMarks>= 50.0)
grade="C";
17.    else grade="D";
18.    return std;
19.    }
20. }//end of class Student
21. // Driver class
22. class Student2Demo{
23. public static void
main(String[] args){
24.    // create an object of Student
25.    Student2 std = new Student2();

```

```

26.    std.SetMarks(97.8);
27.    std.SetName("Ramesh"); // we
have called super method
28.    std.SetIdNo(50595);
29.    //std.salary=19000.00;// error.
salary is private in person2
30.    std.SetSal(30000.00);//ok as we
are using Set method
31.    std=std.ComputeGrade(std);
32.    std.DisplayData();
33.    System.out.println("Details
from Sub class Student: ");
34.    System.out.println("Total
Marks:" + std.GetMarks());
35.    System.out.println("Grade:" +
std.GetGrade());
36.    //System.out.println("Salary"+
std.salary);//Error. salary not visible
37.    System.out.println("Salary : "+
std.GetSal());}
38.    }//end of class Student2Demo

```

**Output:** Person Details are.....

Id Number:50595

Name:Ramesh

Details from Sub class Student:

Total Marks:97.8

Grade:A

Salary : 30000.0

**L** //std.salary=19000.00; // error **Since**  
**i** salary is private and is not visible outside the  
**n** class> **However, Line No 30**  
**e** std.SetSal(30000.00); //ok as we are  
**N** using Set method. The same concept holds good  
**o** for Line No 37as well.

**.  
2  
9  
:**

**L** is calling DisplayData() of super and basic  
**i** data is displayed. However, for display of  
**n** member data belonging to Student2, use is  
**e** made of public accessory methods GetMarks()  
**N** and GetSal(), etc., as shown in Line No 37.

**.  
3  
2  
:**

## 19.4 Using Super Class

In the examples we have covered so far, we have used mutator methods like SetSalary() to set the private member

data. But we know that the best way to initialize the object is by using constructors. A constructor can set the member data and initialize the object. But when inheritance relationship is involved and when we want to create an instance of subclass, we need to constructor of super first, so that super's constructor can make its part of the object and then subclass constructor completes its work of initialization with its part and then only the complete object is ready. Then how to call super's constructor. By simply using super keyword. The next example highlights the usage of keyword super.

### **Example 19.3a: Person2b.java A Super Class for Demonstrating Use of Super**

```
1. package com.oops.chap19;  
2. class Person2b {  
3.    // member data or attributes  
4.    protected String name; //protected
```



```

5. protected int idNo;
6. //Over loaded constructors
7. Person2b()
8. {System.out.println("Super Default
Const called ");
9. name="NoName";idNo=0;}
10.    Person2b(String nm, int id)
11.
{System.out.println("Super (name,idNO)
called");
12.    name=nm;idNo=id;}
13.    Person2b(Person2b per)
14.
{System.out.println("Super (object) Const
called ");
15.    name=per.name;idNo=per.idNo;}
16.    //access and mutator methods
17.    public void DisplayData() {
18.        System.out.println("Person
Details are.....");
19.        System.out.println("Id Number:"
+ idNo);
20.        System.out.println("Name:" +
name);
21.    }
22.    }//end of class Person

```

<b>Lin</b>	are constructors defined by Person2b.
------------	---------------------------------------

**e  
Nos  
. 6,  
9 &  
12:**

These are overloaded constructors and will be called by class that is inheriting from Person2b using the keyword super.

### **Example 19.3b: Student2bDemo.java To Show Calling of Super Constructors**

```
1. package com.oops.chap19;
2. import com.oops.chap19.Person2b;
3. class Student2b extends Person2b {
4.     // member data or attributes
5.     private double totalMarks;
6.     private String grade;
7.     //create object of Student2b by
calling super constructor
8.     Student2b(Student2b std2b) {
9.         super(std2b);
10.         totalMarks=std2b.totalMarks;
11.         grade=std2b.grade;
12.     }
```

```

13.    //constructor with all
parameters given
14.    Student2b(String nm, int rn,
double tm, String gd){
15.        super(nm,rn);
16.        totalMarks=tm;
17.        grade=gd;
18.    }
19.    //access and mutator methods
20.    double GetMarks(){ return
totalMarks;}
21.    void SetMarks(double mks ){
totalMarks=mks;}
22.    String GetGrade(){ return
grade;}
23.    void SetGrade(String gd){
grade=gd;}
24.    //public method
25.    public Student2b
ComputeGrade(Student2b std) {
26.        if( totalMarks >=80.0)
std.grade="A";
27.        else if (totalMarks>= 60.0)
grade="B";
28.        else if (totalMarks>= 50.0)
grade="C";
29.        else grade="D";
30.        return std;
31.    }
32.    public void
DisplayData(Student2b std) {
33.        super.DisplayData();

```

```

34.    System.out.println("Total
Marks:" + std.GetMarks());
35.    System.out.println("Grade:" +
std.GetGrade());
36.    }//end of class Student
37.    }
38.    // Driver class
39.    class Student2bDemo{
40.    public static void
main(String[] args){
41.        // create an object of Student
42.        Student2b std1 = new
Student2b("Ramesh",50595,97.8,"X");
43.        std1.ComputeGrade(std1);
44.        std1.DisplayData(std1);
45.        System.out.println("Total
Marks:" + std1.GetMarks());
46.        System.out.println("Grade:" +
std1.GetGrade());
47.        }
48.    }//end of class Student2bDemo

```

<b>L i n e N o</b>	<p>shows that Student2b extends to Person2b. This means that it has access to all public and protected members like protected String name, protected int idNo and method DisplayData() of Person2b.</p>
--	---

•  
**2**  
:

**L** declare Student2b private data and Line Nos.  
**i** 19–23 define mutator and accessor methods for  
**n** these private variables.  
**e**

**N** Interesting and new points in this program are  
**o** in Line Nos. 8 to 18 that show overloaded  
**s** constructors for Student2b Line No. 8

• Student2b(Student2b std2b) receives  
**4** std2b an object as parameter and calls  
**&** super(std2) in Line No. 9. Remember super of  
**5** Student2b is Person2b and accordingly  
:  
Person(per object) as shown in Line No.  
12 of Example 19.3a will be called. So  
Person2b(per) will set name and id number  
in Line No. 11 and constructor of Student2b  
will set totalMarks and grade in Line No. 10  
and 11 of Example 19.3b.

**L** creates an object of Student2b by supplying  
**i** all the arguments like name, idNo, total  
**n** Marks, and grade. Line No 43 calculates  
**e** grade.  
**N**

•  
**4**

<b>2</b>	
<b>:</b>	
<b>L</b>	calls <code>DisplayData()</code> that in turn calls
<b>i</b>	<code>super.Display()</code> data to display data
<b>n</b>	pertaining to <code>Person2b</code> such as name and
<b>e</b>	<code>idNo</code> as shown at Line Nos. 16–19 (19.3a) and
<b>N</b>	local private variables like <code>totalMarks</code> and
<b>o</b>	grade to accessor methods shown in Line Nos
<b>.</b>	19 and 22.
<b>4</b>	
<b>4</b>	
<b>:</b>	

## 19.5 Methods Overriding

Inheritance relationship allows a subclass to inherit all the protected members of a superclass. Importantly, it can decide to override them and implement its own version. This is like, your daddy has a car. As a descendant, you have the privilege to use Daddy's car or buy a scooter for your exclusive movement. When you call for a transport, obviously you will get a scooter.

**What is overriding:** Superclass method overriding means the same name and return type in the subclass as that defined in the superclass. But the base class will have different implementations. For example, superclass Person defines two methods, namely, `Work()` and `Learn()`:

---

```
public void Learn()  
    { System.out.println("\n Persons  
learn general subjects...");}  
public void Work()  
    { System.out.println("\n Persons work  
at work spots.....");}
```

---

Subclass, if it does not define these functions once again, it can use superclass `learn()` or `Work()`. But if it wants its own implementation, different from base class, it can define these methods in its class definition.

---

```
public void Learn()  
    {System.out.println("\n Students learn  
professional subjects...");}  
public void Work()  
    {System.out.println("\n Students work  
at college laboratories...");}
```

---

When subclass object calls these functions, it will get derived class functionality only but NOT the base class functionality. The concepts are shown in our next example.

### **Example 19.3c: A Program to Show Super Class Methods Overriding by Subclass**

```
1. package com.oops.chap19;
2. public class Person3 {
3.     public void Learn(){
System.out.println("Persons Learn ");}
4.     public void Work()
{System.out.println("Persons work at
work spots");}
5. }//end of class Person3
```

---

**Compile it as Person3.java**

--	--



<b>Line No. 3 &amp; 4:</b>	declare two public methods called Learn() and Work()
--------------------------------	---

## **Student3 Class and StudentDemo To Show Overriding of Super Methods**

```

1. package com.oops.chap19;
2. import com.oops.chap19.Person3;
3. class Student3 extends Person3 {
4.     //public method
5.     public void Learn()
6.     {System.out.println("Students
Learn at college");}
7.     public void Work()
8.     {System.out.println("Students work
at college labs");}
9.     public void Play()
10.    {System.out.println("Students
play at college grounds");}
11.    }//end of class Student3
12.    // Driver class
13.    class StudentDemo3{
14.        public static void
main(String[] args){
15.            // create an object of Student
16.            Student3 std = new Student3();
17.            System.out.println("sub class
calls over ridden methods");
18.            std.Learn();

```

```

19.    std.Work();
20.    System.out.println("sub class
calls its own method");
21.    std.Play();
22.    }
23.    }//end of class Student1Demo

```

**Output:** sub class calls over ridden  
methods

```

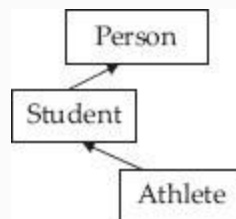
Students Learn at college
Students work at college labs
sub class calls its own method
Students play at college grounds

```

**L** shows that Student3 extends to Person3.  
**i** Accordingly, we have imported Student3 in  
**n** Line No 2. Line No. 9 shows that subclass can  
**e** define its own method called Play() and use it  
 in Line No 21. In Line Nos. 5 and 7, subclass  
**N** Student3 overrides base class methods with its  
**o** own implementations. So when Student3  
**.** object std calls Learn() and Play() in Line  
**3** Nos. 18 and 19, its own methods are called and  
**:** supermethods are overridden.

## 19.6 Multilevel Inheritance

Examples 19.1 to 19.3 have shown single inheritance. We can also have multilevel inheritance. This means that Student can derive from Person and Athlete can then be derived from Student. It is like Grandfather, Father and Son relationship. When such multilevel inheritances are planned, remember that all subclasses can have access to all the superclasses. This means that the Son can have access to protected and public members of both Father and Grandfather. We have shown the multilevel inheritance at Figure 19.2 between the superclass Person and multilevel inheritance as Student and Athlete.



**Figure 19.2** Multilevel inheritance

We have already developed code for Person2b and Student2b . We now want

to create a new class called `Athlete2b` from `Student2b`. This means that we want to create a new class by extending an existing class. There is no need to write or duplicate the code for `Student2b` or `Person2b` again. We can simply import these classes and create a new class by extending, as shown in our next example. This is the power of inheritance and demonstration of the reuse feature of Java.

## Example

### 19.4: `Athlete2bDemo.java`

```
1. package com.oops.chap19;
2. import com.oops.chap19.Student2b;
3. class Athlete2b extends Student2b
{
4. // member data or attributes
5. private String sport;
6. private String level; // S/N/I for
state and N for national
7. //create object of Athlete2b by
```

calling super constructor

```
8. Athlete2b (Athlete2b ath2b) {  
9.   super(ath2b);  
10.   sport=ath2b.sport;  
11.   level=ath2b.level;  
12.   }  
13.   //constructor with all
```

parameters given

```
14.   Athlete2b (String nm, int rn,  
double tm, String gd, String sp,String  
lv) {  
15.   super (nm, rn, tm, gd);  
16.   sport=sp;  
17.   level=lv;  
18.   }  
19.   //access and mutator methods  
20.   public String GetSport() {  
return sport;}  
21.   public String GetLevel() {return  
level;}  
22.   //public method  
23.   public void  
DisplayData(Athlete2b ath2b) {  
24.   super.DisplayData(ath2b);  
25.   System.out.println("Details  
from Athlete2b are :");  
26.   System.out.println("Sport:" +  
ath2b.GetSport());  
27.   System.out.println("Level:" +  
ath2b.GetLevel());  
28.   } //end of class Student  
29.   }
```

```

30.    // Driver class
31.    class Athlete2bDemo{
32.    public static void
main(String[] args){
33.    // create an object of Athlet
34.    Athlete2b ath1 = new Athlete2b
("Ramesh",50595,97.8,"X","FootBall","S")
;
35.    ath1.ComputeGrade(ath1);
36.    ath1.DisplayData(ath1);
37.    }
38.    }//end of class Athlete2bDemo
Output: Super(name,idNO) called
Person Details are..... Id Number:50595
Name:Ramesh
Details from Student2b are : Total
Marks:97.8 Grade:A
Details from Athlete2b are :
Sport:FootBall Level:S

```

<b>Line No. 2:</b>	we are importing class Student2b . Class Student2b in turn imports from Person2 . In Line No. 3, Athlete2b extends to Student2b . Hence, we have access to both Student2b and Person2b .
--------------------	--

<b>Line</b>	declare private variables of athlete2b and
-------------	--

**n  
e  
N  
o  
s.  
5  
&  
6:**

Line Nos. 19 and 21 define accessor methods for private variables.

**Li  
n  
e  
N  
o  
s.  
8  
&  
1  
4:**

are constructors overloaded with object as argument and also all data as arguments. Notice that `Athlet2b` constructor call `super`, i.e. `Student2b` constructor, and that in turn calls `Person2b` constructor thus making the picture (object) complete.

**Li  
n  
e  
N  
o  
s.  
3  
6  
&  
2  
3:**

call `DisplayData()` of `super`, i.e. `Student2b`, by supplying `ath2b` object. `Student2b` in turn calls `Displaydata()` of `Person2b`. Thus, each code segment displays its part of data.

<b>Line No. 35:</b>	computes grade by calling <code>ComputeGrade()</code> method of super class <code>Student2b</code> .
---------------------	--

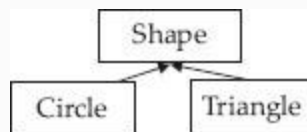
## 19.7 Run-time Polymorphism

Hitherto, what we have shown is overriding of super methods by Java's subclasses. This feature is exploited by Java to give programmers a powerful tool to resolve the method overriding at run time rather than at compile time. A reference made to super can refer to any one of the overridden subclass methods. But which overridden method is to be called? Java simply decides this based on the type of the object referred to at run time.

Therefore, we can say that whichever type of object is referred to by reference to super class, that object's particular overridden method is called. We will explain this with a simple program. Refer to [Figure 19.3](#). There is a super class called `Shape`, and two



subclasses derived from Shape are Circle and Triangle. Super declares a method void FindArea() which is overridden by both Circle and Triangle with their own implementation. Our next example shows how we can use run-time polymorphism feature and call the implementation based on the type of object at run time.



**Figure 19.3** Run time polymorphism

### **Example 19.5: RunTimePoly.java To Demonstrate Run-time Polymorphism Feature of Java.**

---

```
//RunTimePoly.java a programme to  
shown run time ploy morphism
```

```

1. package com.oops.chap19;
2. class Shape{
3. void FindArea(){
4. System.out.println("Inside Shape.
No definite Area");}
5. }// end of Shape
6. class Circle extends Shape{
7. void FindArea(){
8. System.out.println("Inside Circle.
Area=PI*r*r");}
9. }// end of Circle
10. class Triangle extends Shape{
11. void FindArea(){
12. System.out.println("Inside
Triangle. Area=0.5*base*alt");}
13. } // end of class Triangle
14. class RunTimePoly {
15. public static void
main(String[] args) {
16. // create object to classes
17. Shape shp = new
Shape();//object of Shape
18. Circle circ = new
Circle();//object of Circle
19. Triangle trg = new
Triangle();//object of Triangle
20. System.out.println("A ref to
super can refer to sub class over ridden
Methods");
21. System.out.println("Therefore
create a ref to super");
22. Shape refsuper; // reference to

```

```

super class Shape
    23.    System.out.println("A reference
to super ");
    24.    refsuper=shp; // refsuper
refers to Shape object
    25.    refsuper.FindArea();
    26.    System.out.println("A reference
to Circle ");
    27.    refsuper=circ; // refsuper
refers to Circle object
    28.    refsuper.FindArea();
    29.    System.out.println("A reference
to Triangle ");
    30.    refsuper=trg; // refsuper
refers to Triangle object
    31.    refsuper.FindArea();}
    32.    }// end of runtime poly
Output: A reference to super can refer
to sub class over ridden Methods
Therefore create a referecne to super
A reference to super
Inside Shape. No definite Area
A reference to Circle
Inside Circle. Area=PI*r*r
A reference to Triangle
Inside Triangle. Area=0.5*base*alt

```

---

<b>L</b>	create objects to classes Shape, Circle and
----------	---

<b>i</b> <b>n</b> <b>e</b> <b>N</b> <b>o</b> <b>s.</b> <b>1</b> <b>7</b> <b>—</b> <b>1</b> <b>9</b> <b>:</b>	Triangle. Shape is super class and Circle and triangle are subclasses, as shown in Line Nos. 6 and 10. Each subclass Circle and Triangle has overridden super method <code>FindArea()</code> in Line Nos. 8 and 11.
---	---

<b>L</b> <b>i</b> <b>n</b> <b>e</b> <b>N</b> <b>o</b> <b>.</b> <b>2</b> <b>2</b> <b>:</b>	obtains a reference to super class object. Line Nos. 24, 27 and 30 show that reference is made to refer to objects of Shape, Circle and Triangle. Therefore, in Line Nos. 25, 28 and 31, the corresponding overridden methods are called.
--	---

## 19.8 Abstract Classes

When we need to ensure that all subclasses have to share common methods but at the same time have their own implementations,

we use abstract class. Abstract class will have one or more abstract methods. An abstract need not have any abstract method at all.

An abstract method is a method that is declared but not defined. It is declared with the keyword `abstract`, and has a header like any other method, but ends with a semicolon and contains no method body.

The syntax is: `abstract type Method Name (arguments) ;`

A non-abstract class is sometimes called a concrete class.

An abstract class cannot be instantiated because it contains abstract methods. However, subclasses derived from abstract class can have their own implementations. Let us revisit our example at 19.5, but this time we would declare `Shape` as abstract class.

### **Example 19.6: `Abstract.java` A Program to Show Abstract Class and Methods Usage**

```
1. package com.oops.chap19;
2. abstract class Shape1{
3.     abstract void FindArea(); //
abstract method
4.     public void DisplayCommon(){
5.         System.out.println("Inside Shape.
Exiting the Shape1 common message");
6.     }
7. }//end of class Shape1
8. class Circle1 extends Shape1{
9.     void FindArea(){
10.         System.out.println("Inside
Circle1. Area=PI*r*r");
11.         super.DisplayCommon();//
display common message thru super
abstract class
12.     }
13. }// end of Circle1
14. class Triangle1 extends Shape1{
15.     void FindArea(){
16.         System.out.println("Inside
Triangle1. Area=0.5*base*alt");
17.         super.DisplayCommon();
18.     }
19. }//end of Triangle1
20. class AbstractDemo{
21.     public static void
main(String[] args) {
```

```

22.    // create object to classes
23.    //Shape1 shp = new
Shape1();//error.can't instantiate
abstract class
24.    Circle1 circl = new
Circle1();//object of Circle
25.    Triangle1 trgl = new
Triangle1();//object of Triangle
26.    System.out.println("A ref to
super calls sub class over ridden
Methods");
27.    System.out.println("create a
ref to super");
28.    Shape1 refsuper; // reference
to super class Shape
29.    System.out.println("A ref to
super : refsupper is created");
30.    System.out.println("ref to
refer to object of Circle1 ");
31.    refsupper=circl;
32.    refsupper.FindArea();
33.    System.out.println("ref to
refer to object of Triangle ");
34.    refsupper=trgl;
35.    refsupper.FindArea();
36.    }
37.    }

```

**Output:** A refrence to super can refer  
to sub class over ridden Methods

Therefore create a referecne to super

A refrence to super : refsupper is  
created

reference made to refer to object of  
Circle1

Inside Circle1. Area=PI\*r\*r

Inside Shape. Exiting the Shape1

common message

reference made to refer to object of  
Triangle

Inside Triangle1. Area=0.5\*base\*alt

Inside Shape. Exiting the Shape1

common message

**L  
i  
n  
e  
N  
o  
.  
2  
:**

declares an abstract class. It is declared as abstract because it has one abstract method defined in Line No. 3, only name and no body:  
**abstract void FindArea();**

**L  
i  
n  
e  
N  
o  
.**

declares a non-abstract method: **public void DisplayCommon()** that is used to implement a common functionality to all derived subclasses. What is common to all, it is better to implement it at abstract class level instead of at each subclass.



**4  
:**

**L** declare and define concrete methods in classes  
**i** Circle1 and Triangle1 . Note how at the  
**n** end of specialized method called FindArea()  
**e** these methods are invoking common  
**N** functionality, an exit routine we can say by  
**o** calling super.DisplayCommon() ; in Line  
**s** Nos. 11 and 17.

**.  
8  
t  
o  
1  
9  
:**

Refsuper is a reference created in Line No. 28 and it is made to refer to object of Circle1 and Triangle1 in Line Nos. 31 and 34.

## 19.9 Using Final with Inheritance

In the previous chapter, we have shown how final keyword can be used to define variables

and maintain their values constant and unchanged during the running of the program.

### *19.9.1 Final Method*

Final qualifier can be used with methods also. You are aware that in inheritance, the derived class can override base class methods with its own implementation. When we do not want such a thing to happen, we need to declare the method in base class as final. For example:

#### **Example 19.7: Java Code Segment to Show the Use of Final Qualifier for a Method. Constructors and Destructors of a Class**

```
class Person {  
    final void Update() { //update code  
        here. Final Methods. Cannot be  
        inherited}  
} // end of person
```

```
class Student extends Person {  
    void update() { // Error cannot be  
inherited}  
} // end of student
```

---

### *19.9.2 Final Classes*

When we do not want a class to be inherited we declare a class as final. As class cannot be inherited, then automatically by default all the members inside a final class are considered as final and hence cannot be inherited.

#### **Example 19.8: Java Code Segment to Show the Use of Final Qualifier for a Class. Constructors and Destructors of a Class**

---

```
final class Person {  
    void Update() { //update code here.  
Final Methods. Can not be inherited}  
} // end of person  
class Student extends Person { //
```

```
error Student cannot extend to
Person(final class)
    Void update() { // Error cannot be
inherited}
    } // end of student
```

---

## 19.10 Object Class

By now, we appreciate the power of run-time polymorphism and reuse of code already written. The objects we create are automatically treated as subclasses for the class called Object in java.lang package. What is the advantage gained by this?

The Object class contains several useful static methods that can be used directly without creating object. Now that all object user creates are treated as subclasses, all user-created objects will get access to the static method in meta class Object in package java.lang. The methods with final specifier cannot be overridden. Methods supported by the object class are given in Table 19.1.

**Table 19.1** Static and final methods of object class

Static Method	Remarks
---------------	---------

boolean equals()	Compares the two objects by checking if reference is same. If same returns true, else returns false.
String toString()	Returns a String describing the object. It is automatically invoked along with println statement used with object.
int hashCode()	Returns hash code linked to objects invoking.
void finalize()	Called before clearing of object no longer required by Garbage Collector.
final void notify()/notifyAll()	Resumes highest priority thread/all thread waiting on the invoking object.
Object clone()	Creates a clone, i.e. a new object that is same as that of object being cloned.
final void wait()/wait(ms)/wait(ms,ns)	Waits on specified thread (not under execution) for specified time.
final Class getClass()	Obtains class name of object at run time.

## Example 19.9: ObjectDemo.java A Java Program to Show the Use of Static Method of Object Class

```
1. package com.oops.chap19;
2. class Person4 implements
Cloneable{
3.   String name;
4.   Person4(String nm) {name=nm;}
5.   public String Getname() {return
name;}
6.   public void DisplayData() {
7.     System.out.println("Person Details
are.....");
8.     System.out.println("overriden
toString() Method:" + toString());
9.     System.out.println("Obj Details
from static method hashCode()");
10.    System.out.println("Object Hash
Code :" + hashCode());}
11.    public String toString()
12.    {String stg;
13.     stg = "Name :" + Getname();
14.     return stg;}
15.    public void
CompareObjects(Person4 per2){
```

```

16.    System.out.println(" using
equals(obj) static method");
17.    if (this.equals(per2))
18.    System.out.println("Per1& Per2
are same.");
19.    else System.out.println("Per1 &
Per2 are NOT same.");}
20.    public Object CloneObj() throws
CloneNotSupportedException
21.    {return super.clone();}
22.    }//end of class Person
23.    public class ObjectDemo {
24.    public static void
main(String[] args) throws
CloneNotSupportedException
25.    {
26.    //Two different objects. Their
references are NOT same
27.    // But Content same
28.    System.out.println("Two objects
per1&2 with same content");
29.    Person4 per1 = new
Person4("Salman");
30.    Person4 per2 = new
Person4("Salman");
31.    per1.CompareObjects(per2);
32.    System.out.println("Displaying
per1 data");
33.    per1.DisplayData();
34.    System.out.println("Clone per1
object to per3");
35.    Person4 per3 =

```

```

(Person4)per1.CloneObj();
36.    System.out.println("Displaying
per3 data");
37.    per3.DisplayData();
38.    }
39.    }

```

**Output:** Creating two objects per1 and per2 with same content using equals(obj) static method

```

Perl & Per2 are NOT same.
Displaying per1 data
Person Details are.....
Using overridden toString()
Method:Name :Salman
Object Details from static method
hashCode() of Object class are.....
Hash Code of Object:1671711
Clone per1 object to per3
Displaying per3 data
Person Details are.....
Using overridden toString()
Method:Name :Salman
Object Details from static method
hashCode() of Object class are.....
Hash Code of Object:11394033

```

<b>L</b> <b>i</b>	declares a class Person4 implementing an interface called Cloneable. Cloneable is called
----------------------	--



**n** interface. More details about the interface in  
**e** succeeding sections. All the methods described  
**N** by interface are available to `Person4`.  
**o**  
**.**  
**2**  
**:**

**L** is a constructor that assigns data to name as per  
**i** argument supplied.  
**n**  
**e**  
**N**  
**o**  
**.**  
**4**  
**:**

**L** overrides the static method `toString()`  
**i** method. Hence when you call `toString()` as  
**n** in Line No. 8, own overridden method is called.  
**e**  
**N**  
**o**  
**.**  
**1**  
**o**  
**:**

**L** defines the overridden `toString()` method.  
**i** Line No. 15 is about Object  
**n** `CompareObject(Person4 obj)` that takes

**e** in object and compares the argument object  
**N** with object that is invoking the object. It checks  
**o** the reference of both objects and if they are  
**.** equal, returns true and else returns false.  
**1**  
**1**  
**:**

**L** **public Object CloneObj () throws**  
**i** CloneNotSupportedException is a public  
**n** method CloneObj () that returns clone using  
**e** clone () method of Object class.  
**N**  
**o**  
**.**  
**1**  
**9**  
**:**

**L** **throws Clone Not Supported Exception Line**  
**i** No. 17 this refers to object that has invoked the  
**n** method CompareObjects ()  
**e**  
**N**  
**o**  
**.**  
**1**  
**9**  
**&**  
**2**  
**4**  
**:**

## 19.11 Packages

As a developer, you would like to keep all your work together so that you can reuse them in future if needed. We would like to keep all our work as a “package”. Suppose you are working in a company called “oops.com”, then it is customary to create package as “com.oops”. Now we are writing programs for Chapter 19. Hence, we will name our package as “com.oops.chap19”. Refer to Figure 15.7. Path variable will link up to `c:\Oopajava\example\src`. It is your package statement, **package** `com.oops.chap19`; which will link to `com.oops.chap19`.

So what is a package? A package is collection of all related class files into a group so that they can be reused by importing them into other classes and thus aid in enhancing

- Ease of handling complex project development

- Develop reusable classes and components

### *19.11.1 Reusable Classes*

The concepts and commands to be used for package and path and class path have been dealt in detail in Chapter 15 and Section 15.5. The user is strongly advised to go through the concepts and examples to get practice. Here, we concentrate only on amplification of a few relevant concepts like reusability, etc.

- Add suitable package name at the top of the class :  
`package com.oops.chap19.`
- Save the source file in appropriate package directory like  
`C:\Oopsjava\examples\src\com\oops\chap19\`  
`edit Date1.java`. Note that Oopsjava is a directory. Examples is a workspace and src directory to store all our source files. The package name `com.oops.chap19` requires that our source code be placed in  
`C:\Oopsjava\examples\src\com\oops\chap19.`  
This placement is done automatically if your development platform is Eclipse, etc.
- Declare the class as public so that it is accessible by all other classes.
- Compile the class and place it in an appropriate directory. For example:  
`C:\OopsJava\examples\src\com\oops\chap19>`  
`javac -d c:\Oopsjava\examples\bin`  
`Date1.java`

Now go to bin directory and type the command to execute java program giving full package path.

---

```
C:\OopsJava\examples\bin>java  
com/oops/chap19/Date1
```

---

Note that since it's a simple program, we have executed it from bin directory. For large programs or programs with several classes, we would write a driver program and place the class program in directory `TestDriver`. Then we need to execute java command from `TestDriver` directory. Chapter 15 gives you details of setting the path and class path.

## 19.12 Path and Classpath

Chapter 15 (Section 15.6.2) gives full details and settings required for path and class path variables. Setting path is essential if we want to run java executables from any directory without stating full path name each time. Class path, on the other hand, is a message

from you to Java environment, where to search for your class files.

### 19.13 Importing of Packages

Package means grouping all relevant classes so that they can be reused. Therefore, all the class files in the same package are public up to package level and hence need not be imported. However, class files belonging to outside packages are required to be imported. For example, we need to import `java.util.*`; to get access to class file in package `java.util`.

#### **Example 19.10a: Date1.java A Program to Develop Reusable Class Date1 Using Package and Import of Class**

---

```
/* Date1.java Date1 class shows date
in dd:mm:yy format & converts to
dd:mmm:yy format*/
```

```

1. package com.oops.chap19;
2. public class Date1 {
3.     private int dd;
4.     private int mm;
5.     private int yy;
6.     //constructor for Date1
7.     Date1(int d, int m,int y)
{dd=d;mm=m;yy=y;}
8.     //validate date
9.     public void validateDate(int d,int
m,int y)
10.         {dd= ((d>=0 && d<=31 )?d:0);
11.         mm= ((m>=0 && d<=12 )?m:0);
12.         yy= ((y>=1900 && d<=1999)?y:0
);
13.     }// end of validateDate()
14.     // change date format to dd-
mmm-1910 Ex 01:Jan:10 format
15.     public String ConverttoMMM(int
mm){
16.         String mmm="";
17.         switch(mm)
18.         { case 1 : mmm="JAN";break;case
2 : mmm="FEB";break;
19.         case 3 : mmm="MAR";break;case 4
: mmm="APR";break;
20.         case 5 : mmm="MAY";break;case 6
: mmm="JUN";break;
21.         case 7 : mmm="JUL";break;case 8
: mmm="AUG";break;
22.         case 9 : mmm="SEP";break;case
10 : mmm="OCT";break;

```

```

23.    case 11: mmm="NOV";break;case
12 : mmm="DEC";break;
24.    }
25.    return mmm;
26.    }
27.    public String toDDMMYYString()
28.    {return
String.format("%d:%s:%d",dd,ConverttoMMM
(mm),yy);}
29.    public String toString()
30.    {return
String.format("%d:%d:%d",dd,mm,yy);}
31.    }// end of Date1

```

---

<b>L i n e  N o . 1 :</b>	declares package as <code>com.oops.chap19;</code>
---	---

<b>L i n</b>	shows that any reusable class is to be declared as public. Line Nos. 3–5 declare private variables.
----------------------	---



**e**

**N**

**o**

**.**

**2**

**:**

**L**

**i**

**n**

**e**

**N**

**o**

**.**

**6**

**:**

**L**

**i**

**n**

**e**

**N**

**o**

**s**

**.**

**1**

**5**

**-**

**2**

declares a constructor.. Line Nos. 9–13 validate the month, day and year

define a public method **public** String ConverttoMMM(int mm) to convert Date from dd:mm:yy format to dd:mmm:yy format

6  
:

**L** **public** String toDDMMYYString()  
**i** converts to dd:mm:yy format. For this,  
**n** it uses a method ConverttoMMM(mm) in Line  
**e** No. 27. Also note that the method  
**N** toDDMMYYString() uses String.format  
**o** method by specifying the format to make a  
**.** string. Here c like formatting is used. %d for  
**2** integers and %s for strings are used.  
**7**  
**:**

## Example 19.10b: Datetester.java A Driver Program to Test Date1 at Example 19.10a

```
1. package com.oops.chap19;  
   //import com.oops.chap19.Date1;  
Need not be imported because same  
package
```

```

2. public class Datetester{
3. public static void main(String[]
args) {
4. // intialize the Date1 object
5. Date1 date = new Date1(19,11,10);
6. System.out.println("date
in<dd:mm:yy>format"+ date.toString());
7. System.out.println();
8. System.out.println("date
in<dd:mm:yy>format"+ date.toString());
9. System.out.println("date
in<dd:mmm:yy>format"+ date.toDDMMYY
String());
10. }
11. }//end of Datetester class
Output : The Given Date is .....
date in<dd:mm:yy>format19:11:10
date in<dd:mmm:yy>format19:NOV:10

```

---

Both Datetester and date1 belong to the same package. Hence, Date1 need not be imported. Hence, we have commented it out.

<b>Line No. 2:</b>	delcares our driver class Datetester

<b>Line No. 4:</b>	creates an object date of Date1 by calling the constructor of Date1 . This is how we will achieve reusability.
<b>Line Nos. 8 &amp; 9:</b>	call methods date.toString() & date.toDDMMYYString() of Date1 to display date in two different formats.

## 19.14 Access Specifiers Revisited for Packages

The access specifiers are public, private, protected and default also called friendly, i.e. public up to package level. These specifiers act on classes, subclasses and packages. Java allows a special definition called private protected to give visibility to all subclasses ,wherever they are present independent of packages. The visibility is specified in Table 19.2. OK stands for visible. NO stand for not visible.

**Table 19.2** Access privileges

Private	Public	Protected	Package (default)	Private	Protected
Same class	OK	OK	OK	OK	OK
Sub class but same package	OK	OK	OK	OK	NO
Other classes but same package	OK	OK	OK	NO	NO
Sub classes in other package	OK	OK	NO	NO	NO
Classes in other packages	OK	NO	NO	NO	NO

Note that Java allows a special access specifier in private protected, when we want to provide access to subclass anywhere, independent of package.

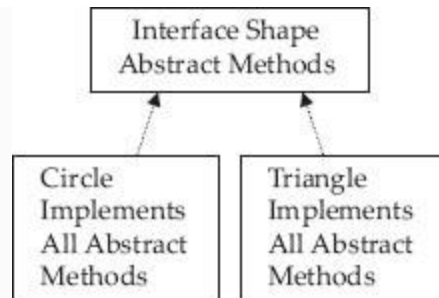
## 19.15 Interfaces

In Section 19.8, we have shown you the power of dynamic polymorphism offered by java. It provided a facility wherein a reference to super can refer to any one of the subclass overridden methods based on the type of object. We have further dealt with abstract classes that contain abstract methods. Abstract methods will have only the head terminated by a semicolon, but no body showing the implementation. We have also learnt that an abstract class cannot be instantiated since it has abstract methods. However, a reference to abstract class can be

created that can then be made to refer to objects of subclasses.

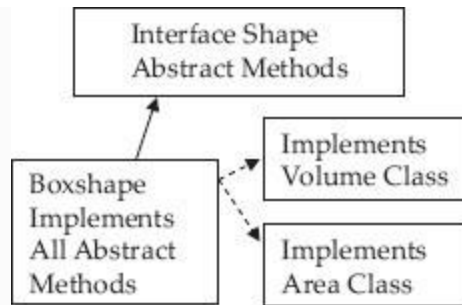
Java permits only single inheritance and multilevel inheritance. A subclass can have only one super. Then how will java solve the problems wherein more than one super is required? The solution is a powerful tool provided by Java called interface. A class in Java can extend only to one super, but it can implement several Interfaces.

Refer to Figure 19.4. It shows a situation wherein super, called interface rather than a class, has abstract methods. Subclasses called Circle and Triangle implement all abstract methods. Also note that the dotted arrow pointing to interface rather than a solid arrow. This is notation as per modelling language UML.



**Figure 19.4** Interface in Java

A second feature of interface is that it solves ambiguity problems associated with inheriting from more than one super. Java does not support multiple inheritances, meaning that it does not support extending to more than one super at a time to avoid ambiguities in Inheritance relationships. This problem is solved by Java by allowing multiple implementations. A class can extend to only one super, but it can implement several interfaces. This situation is shown in [Figure 19.5](#). The syntax is:



**Figure 19.5** Box class extends to shape, but can implement several interfaces

---

```

class Class extends Superclass
implementsInterface1, Interface2
Example : class BoxShape
extends Shpe implements Volume, Area
  
```

---

### 19.15.1 What and Why of Interfaces

An interface is declared with the keyword **interface** instead of the keyword **class**. An interface may contain *only* abstract methods and definitions of constants (i.e., *final* variables). The keyword **abstract** before each method is optional.

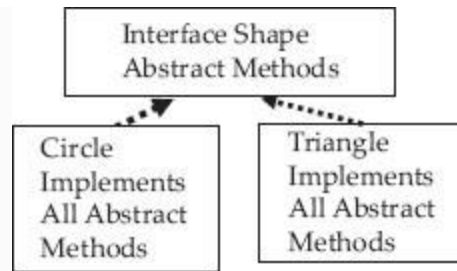
When a class extends an interface, it may implement (define) some or all of the



inherited abstract methods. A class must itself be declared abstract if it inherits abstract methods that it does not define.

### *19.15.2 Defining and Implementing Interfaces*

Interface definition starts with keyword **interface** instead of **class**. Ex **public interface** Shape. Interface can be public or default specifier. Interface can hold only method name and no implementation. This means that all the methods are abstract methods. However, there is no need to use the keyword **abstract** in front of interface. An interface can also define final and static declarations. For example, we show three abstract methods in an interface called Shape in Example 19.11a. The concept diagram is shown in Figure 19.6.



**Figure 19.6** Circle and triangle classes implement shape interface

## Example 19.11a: Shape . java

### Interface Shape With Only Abstract Methods

```
1. package com.oops.chap19;
2. public interface Shape {
3.     final static double PI = 3.141519;
4.     // abstract methods. Only Head no
body.
5.     //Using key word abstract is
optional
6.     double ComputeArea(double x);
7.     double ComputePerimeter(double x);
```

```
8. void DrawShape();  
} // end of interface shape
```

---

## **Example 19.11b: ShapeTest.java Circle and Square classes implement interface**

```
1. package com.oops.chap19;  
2. // Circle  
3. class Circle2 implements Shape  
4. {public double ComputeArea(double  
radius){  
5. double area=0; area =  
PI*radius*radius;  
6. return area;  
7. }  
8. public double  
ComputePerimeter(double radius){  
9. double perim=0;perim =  
2*PI*radius;return perim;}  
10. public void DrawShape(){  
11. System.out.println("Draw Circle  
Shape here!");}  
12. } // end of class Circle1
```

```

13.    class Square implements Shape
14.    {public double
ComputeArea(double side){
15.        double area=0; area =
side*side;return area;}
16.        public double
ComputePerimeter(double side){
17.            double perim=0;perim =
4*side;return perim;}
18.        public void DrawShape(){
19.            System.out.println("Draw Square
Shape here!"); }
20.    }// end of class Square
21.    class ShapeTest{
22.        public static void
main(String[] args){
23.            //Create a reference to
Interface Shape
24.            Shape shp;
25.            //Create objects of sub classes
26.            Circle2 circ = new Circle2();
27.            Square sqr = new Square();
28.            // make ref to refer to Circle
29.            shp=circ;
30.            System.out.println("Area of
Circle"+ shp.ComputeArea(10.0));
31.            System.out.println("Perim of
Circle"+ shp.ComputePerimeter (10.0));
32.            shp.DrawShape();
33.            System.out.println(); //blank
line
34.            shp=sqr;

```

```

35.    System.out.println("Area of
Square"+ shp.ComputeArea(10.0));
36.    System.out.println("Perim of
Square"+ shp.ComputePerimeter (10.0));
37.    shp.DrawShape();
38.    }
39.    }

```

---

As interface Shape and implementing classes belong to the same package, there is no need to import the Shape interface.

**L  
i  
n  
e  
N  
o  
.  
3  
:**

**class Circle2 implements** Shape shows that Circle2 implements Shape. Line Nos. 4–7 show that Circle2 implements abstract method of Shape. Similarly, Line Nos. 8 to 12 show that Circle2 implements two other abstract methods `Computeperimeter()` and `DrawShape()`.

**L  
i  
n  
e  
N  
o**

show the implementation details of Square class.

**S  
.  
1  
3  
t  
o  
1  
9  
:**

**L  
i  
n  
e  
N  
o  
.  
2  
4  
:**

Shape `shp`; creates a reference to interface.  
Line No. 29 makes `shp` to refer to `circ` object.  
Line Nos. 30, 31 and 32 show that `shp` calls the  
solid methods of `circle2` and executes them.

**L  
i  
n  
e  
N  
o  
.  
3  
:**

makes `shp` to refer to object of Square. The  
succeeding lines show implementation of  
Square methods.

## 19.16 Summary

1. Inheritance is a facility provided to create a new class from existing classes. Existing class is called super and new class is called subclass.
2. The access specifiers allowed for members are public, private, protected, private protected, and default access that is public up to package level.
3. If we declare the members as private, then they are visible inside a class only. They are not visible to subclasses or outside the class.
4. Protected is an access specifier that allows subclass to derive members by subclass. Protected is visible to own class and subclass and private to others.
5. Private Protected means that it is visible to any subclass independent of package, even if they do not belong to the same package.
6. Set methods are called mutator methods and Get methods are called accessor methods.
7. Super's constructor can be called by using `super()`
8. Super classes method overriding means the same name and return type in the subclass as that defined in super class. But base class will have different implementations.
9. Java allows only single inheritance and multilevel inheritances. Java does not support multiple inheritances.
10. Abstract Method contains only heading and no body. An abstract class is one which contains abstract methods. Since abstract class has abstract methods, object cannot be instantiated.
11. A reference can be created to super abstract class and this reference can be made to refer to any one of the subclass objects at run time. Thus, a reference can refer to any one of the subclass methods at run time. This is called run time polymorphism.

12. A non-abstract class is called concrete class.
13. A method declared as final cannot be overridden by subclass.
14. A class declared as final cannot be inherited.
15. A variable declared as final cannot change its value during running of the program.
16. java.util package contains a class called Object class that defines static methods. All the user-created objects are treated as descendants of Object class. Hence, all static methods declared in Object class can be directly called by all objects.
17. Package is a collection of all related classes in a single directory with a view to afford reusability of classes.
18. Setting path affords using of java executables like exe files from any directory without the need to specify complete the path every time.
19. Setting class path is an indication as to where to find classes. Dot ( . ) means current working directory.
20. Importing is carried out by import statement to facilitate using the externally defined class. The classes in the same package need not be imported.
21. Interface is a powerful tool provided by Java for dynamic polymorphism. An interface class uses a keyword interface rather than a class.

## Exercise Questions

### Objective Questions

1. Members that can be inherited in Java are
  1. public
  2. private
  3. private protected
  4. protected
  5. default



1. i and ii
2. i and iii
3. i, iii, iv and v
4. ii, iv and iv

## 2. Super class constructor can be called by using

1. `super()`
2. `super.Class Name`
3. `Class Name`
4. `super(Class Name)`

## 3. Java supports the following types of inheritance relationships:

1. Multiple
2. Multilevel
3. Single Level
4. Virtual

1. i and iv
2. i, ii and iii
3. ii and iii
4. iii and iv

## 4. Which of the following statements are true in respect of calling constructors by objects of subclass?

1. Subclass constructor followed by super
2. Super constructor
3. Sub class
4. Super followed by subclass constructor

## 5. Overriding of methods means

1. Same name and same return type as that of super by subclass.
2. Defining more than one method with the same name but different arguments by subclass.
3. Defining more than one method with the same name but different arguments by super.
4. Defining more than one method with the same name and arguments by subclass.

## 6. Which of the following statements are true in respect of abstract classes?

1. Contains all abstract methods only
2. Can contain non-abstract method
3. Can contain NIL abstract methods
4. Object can be instantiated

1. ii, iii and iv
2. i, ii and iii
3. i, ii, iii and iv
4. ii, iii and iv

7. Which of the following statements are true in respect of run-time polymorphism provided by Java?

1. A reference can be created to abstract class.
2. Calling subclass methods by reference is based on type of reference.
3. Calling subclass methods by reference is based on type of Object.
4. Reference to super can refer to any one of the methods of subclasses.

1. ii, iii and iv
2. i, ii and iii
3. i, ii, iii and iv
4. iii and iv

8. Which of the following statements are true in respect of abstract classes?

1. A reference can be created to abstract class.
2. Abstract classes are also called concrete classes.
3. Calling subclasses methods by reference is based on type of Object.
4. Reference to abstract class can refer to any one of the methods of subclasses.

1. i, iii and iv
2. i, ii and iii
3. i, ii, iii and iv
4. i, ii and iv

9. Which of the following statements are true in respect of final keyword?

1. Final variables can change its values in subclasses.
2. Final methods can be inherited.
3. Final methods cannot be overridden.
4. Final classes can be inherited.

1. i, iii and iv
2. ii and iii
3. i, ii, iii and iv
4. i, ii and iv

10. Which of the following statements are true in respect of Object class?

1. It is a super class for all user-created objects.
2. final method declared in Object class can be overridden by user objects.
3. All methods of Object class are static methods.
4. Methods of Object class can be called directly by user-created objects.

1. i, iii and iv
2. ii and iii
3. i, ii, iii and iv
4. i, ii and iv

11. Which of the following statements are true in respect of Interface of Java?

1. It is super class with all member methods as abstract methods.
2. It can contain non-abstract methods.
3. Interface cannot be instantiated but a reference can be created.
4. Reference to interface can call any object methods.

1. i, iii and iv
2. ii and iii
3. i, ii, iii and iv
4. i, iii and iv

12. Interface can contain final static variable declarations TRUE/FALSE

#### Short-answer Questions

13. What are the types of inheritance supported by Java?
14. Distinguish private and protected members.
15. Distinguish protected and private protected access specifiers.
16. What is a super class?
17. What is method overriding?
18. What is a final method?

19. What is an Object class?
20. What is final class?
21. Explain the abstract class.
22. What is run-time polymorphism?
23. What is an Interface?

#### **Long-answer Questions**

24. Explain the access specifiers in Java with suitable examples.
25. Explain the uses of super class with examples.
26. Explain how method overriding helps in achieving run-time polymorphism.
27. How does Java solve the problem of multiple inheritance, considering that Java does not support multiple inheritance.
28. Explain run-time polymorphism feature of Java.
29. Distinguish abstract class and interface.
30. Explain the uses of Final with respect to inheritance.
31. Explain how Object class is supporting reusability feature.
32. What is a package? Provide the syntax for setting path and CLASSPATH. State the Directories being used.
33. Explain why Interfaces are required and their contribution to reusability feature by Java.

#### **Assignment Questions**

34. Explain the role of abstract classes in achieving polymorphism in Java. Explain how polymorphism helps to achieve reusability of code.
35. Discuss the environment variables settings and package concept to achieve the reusability of classes developed. Comment specifically why we have to choose a particular access specifier like public or default, etc.
36. Distinguish the abstract class and Interface. When the former is achieving the polymorphism why are interfaces further developed?

37. Implement an interface called `PayOut` with methods like `GetData()` `ComputeCredits()` and `GetDebits()`, etc. Let class `Employee` implement the interface `PayBill`. Assume necessary fields.
38. Let a class called `Invoice` implement two interfaces called `Quotation` with methods like `DisplayItem()` to display quotation details such as cost, number of items and total cost. `Invoice` also implements interface called `PayOut` at Problem 4 to decide the amount payable to the supplier.

### **Solutions to Objective Questions**

1. c
2. a
3. c
4. d
5. a
6. b
7. d
8. a
9. b
10. a
11. d
12. True

# 20

## Errors and Exceptions in Java and Multithreaded Programming

### LEARNING OBJECTIVES

*At the end of this chapter, you will be able to understand and use*

- Errors and exceptions of Java.
- How and where they are generated and how they are handled.
- Create your own exception class.
- Understand the underlying concepts in multithreaded programs.
- Understand the problems of racing and solution of synchronization of methods.

### 20.1 Introduction

Many of you would have come across a pop-up while you are using the Internet or operating system, wherein the pop message would say “OS or Browser has experienced a serious problem and needs to be closed down. Error reporting is in process”. Indeed, an error has occurred. Despite elaborate testing prior to delivery, errors cannot be prevented, but they can be minimized. In this chapter, we will study the mechanism of Java to handle errors and exceptions.

An important and innovative feature of Java programming is multithreaded programming. Just as Operating System manages several processes concurrently and schedules the next process CPU should handle, Java provides multithreaded programming wherein we can create several concurrent threads and manage thread scheduling at programmer's level. The advantage is that more than one thread process can be running at the same time albeit with different priorities.

## 20.2 Errors and Exceptions

## 20.2.1 Errors

Errors that crop up in a program are of three types:

- **Syntactical Errors:** Can be easily detected and corrected by compilers.
- **Logical Errors:** Arise due to the programmer not understanding flow of logic. This can be corrected by extensive testing.
- **Bugs:** These can be fixed by the programmer using `assert()` macros and debuggers available.

## 20.2.2 Exceptions

Exceptions, on the other hand, are unusual conditions that occur at run time and can lead to errors that can crash a program. The exceptions can be classified as:

### 20.2.2.1 Synchronous

Synchronous are those that can be predicted. For example, array index going outside the permissible values can be easily detected because such errors occur only when you have executed the statement involving array index. The synchronous exception can occur at any one or more of the following situations:



- Memory allocation exception.
- IO handling exception.
- Maths handling exception like division by zero.

#### **20.2.2.2 Asynchronous Exceptions**

Asynchronous exceptions are those that cannot be predicted. Generally, the resources required for the program are allocated at the very beginning of the program and resources are demanded at run time. Thus, it is likely that our program may exceed the initial allotment. As an example, consider allocated memory for an array. When such an exception occurs, we need a mechanism to carry such information to an area where resources are allocated so that we can take corrective actions and prevent the program from crashing.

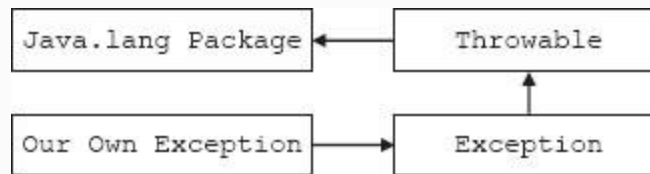
#### **20.2.2.3 Checked Exceptions**

The exceptions that are checked by compiler are called checked exceptions. Example:

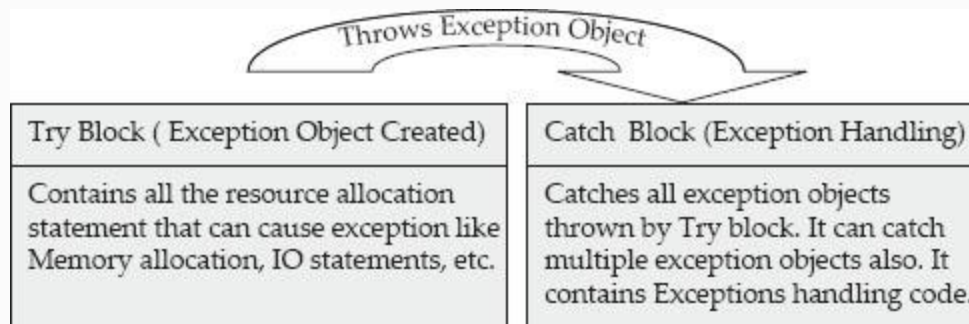
```
public static void main(String  
[] args) throws IOException.
```

#### **20.2.2.4 Unchecked Exceptions**

The exceptions checked by JVM at run time are called unchecked exceptions. All exceptions and errors are objects in Java. These objects are derived from the package `java.lang.Throwable` shown in Figure 20.1.



**Figure 20.1** Exception class hierarchy



**Figure 20.2** Try and catch blocks in exception handling mechanism

#### 20.2.2.5 Exceptions in Java

The most likely area for exceptions to occur is where resources are allocated or where input and output programming is involved. These are listed in Table 20.1.

**Table 20.1** Java exceptions

Java Exception	Remarks
Arithmetic Exceptions	Divide by zero and other mathematical exceptions
ArrayIndexOutOfBoundsException	Index going out of originally defined dimension
FileNotFoundException	File not existing
IOException	Caused by IO statements
NullPointerException	Reference to a null object
NumberFormatException	Occurs when conversion from number to String fails
OutOfMemoryException	Memory limits exceeded
StackOverflowException	Stack over flow
StringOutOfBoundsException	Occurs when limit of String definition is exceeded

## 20.3 Try and Catch Blocks

Java raises an exception object. For it to do so, we need to use a try block wherever we expect likely exception. Catch block that follows the try block catches the exception object and takes remedial action.

### **Example 20.1: Use of Try and Catch Block. Allocation of Memory for a Two-dimensional Matrix**

```
1. package com.oops.chap20;
2. import java.util.*;
3. class MatrixAlloc {
4. void ReadMat(int A[][],int rows,
int cols){
5. //input matrix related data using
scanner
6. Scanner scn=new
Scanner(System.in);
7. for ( int i=0;i<rows;i++)
8. for ( int j=0;j<cols;j++)
9. A[i][j]=scn.nextInt();
10. }
11. void DisplayMatrix(int A[][],int
rows, int cols){
```

```

12. for ( int i=0;i<rows;i++)
13. {for ( int j=0;j<cols;j++)
14. { System.out.print( A[i][j] +"
");}
15. System.out.print( "\n");}
16. }//end of DisplayMatrix
17. void AddMat(int A[][],int B[]
[],int rows, int cols){
18. for ( int i=0;i<rows;i++)
19. {for ( int j=0;j<cols;j++)
20. System.out.print( (A[i][j]+B[i]
[j]) +" ");
21. System.out.print( "\n");}
22. }//end of TranspMat
23. }//end of MatrixAlloc
24. class TwoDMatrix{
25. public static void main(String[]
args)
26. {try{
27. Scanner scn = new
Scanner(System.in);
28. MatrixAlloc obj=new
MatrixAlloc();
29. System.out.print("Enter no of
rows & columns:");
30. int rows= scn.nextInt();
31. int cols= scn.nextInt();
32. int A[][]= new int[rows][cols];
33. int B[][]= new int[rows][cols];
34. MatrixAlloc mat = new
MatrixAlloc();
35. System.out.println("Enter Data

```

```

for Matrix A");
    36. obj.ReadMat( A ,rows,cols);
    37. System.out.println("Enter Data
for Matrix B");
    38. obj.ReadMat( B ,rows,cols);
    39. System.out.println("Addition of
two Matrices");
    40. mat.AddMat(A,B,rows, cols);
    41. } catch( Exception e){};
    42. }
    43. } //end of MatTranspose
OUTPUT : Enter no of rows & columns:2
2
Enter Data for Matrix A 10 20 30 40
Enter Data for Matrix B 10 20 30 40
Addition of two Matrices
20 40
60 80

```

**L** shows that we have included try { block which  
**i** will extend up to Line No. 41 which is catch  
**n** block. We have put a try block because we have  
**e** memory allocation at Line Nos. 27 & 28 :

Scanner scn = new  
**N** Scanner(System.in);MatrixAlloc  
**o** obj=new MatrixAlloc(); Line No. 32 & 33  
**.** allocate memory to matrices A and B.  
**2**

**6**  
**:**

**L** catches Exception e. Of course no further action  
**i** is planned as indicated by { }  
**n**  
**e**

**N**  
**o**  
**.**  
**4**  
**1**  
**:**

In our next example, we will show  
Arithmetic type of exception called divide by  
zero exception.

### **Example 20.2: DiveByZero.java A Program to Show Divide by Zero Exception**



---

```
1. package com.oops.chap20;
2. import java.util.*;
3. public class DivideByZero {
4. public static void main(String[]
args) {
5. double
mass,radius,height,area,density=0.0;
6. boolean yesno=true;
7. Scanner scn = new
Scanner(System.in);
8. System.out.println("Enter mass of
cylinder");
9. mass=scn.nextDouble();
10. System.out.println("Enter height
of cylinder ");
11. height=scn.nextDouble();
12. System.out.println(" Enter radius
of cylinder<0 to test divide by zero and
exit");
13. radius=scn.nextDouble();
14. while(yesno){
15. try
16. { if( radius ==0.0) throw new
ArithmeticException("Divide By Zero");
17. else
18. { area = 2*(22/7)*radius*
(radius+height);
19. density=mass/area;
20. System.out.println("\n Density :
"+ density);
21. System.out.println(" Enter radius
```

```

of cylinder<0 to test divide by zero");
22. radius=scn.nextDouble();
23. }
24. }catch(ArithmeticException e)
25. {System.out.println("\n Divide By
Zero Exception Occured"); yesno=false;}
26. }// end of while
27. }// end of main
28. }// end of DivideByZero
OUTPUT : Enter mass of cylinder 200
Enter height of cylinder 20
Enter radius of cylinder<0 to test
divide by zero and exit 10
Density : 0.11111111111111111
Enter radius of cylinder<0 to test
divide by zero and exit0
Divide By Zero Exception Occured
Exiting the program

```

<b>L i n e N o s . 8</b>	obtains the data for finding out density of cylinder using the formula $\text{density} = \text{mass}/\text{area}$ at Line No. 19
--	--

**t  
o  
1  
3  
:**

**L** is a try block and Line No. 23 is a corresponding  
**i** catch block that catches  
**n** `ArithmeticDivideByZero` exception. We  
**e** are simply informing the user and exiting the  
**N** while loop and programme by making  
**o** `yesno=false`.

**.  
1  
5  
:**

**L** { `if( radius ==0.0)` throw new  
**i** `ArithmeticException ("Divide By`  
**n** `Zero");` throw an exception when radius is  
**e** zero. Thereafter catch block catches the  
**N** exception. Note that we are using the  
**o** parameterized constructor of  
**.** `ArithmeticException` class. More  
**1** importantly it is us who are throwing the  
**6** exception by calling new new  
**:** `ArithmeticException()` parameterized  
constructor.

## 20.4 Handling of Multiple Exceptions by Try and Catch Blocks

A try block can be made to initiate several exception objects. Accordingly catch blocks can catch multiple exception objects. The syntax and an example is shown in the next example.

**Example 20.3: A Try block can Catch Multiple Exceptions Like IOException, Index Out of Bouns Exception, etc. A Catch Block can Catch Multiple Exceptions Like IOException , IndexOutOfBoundsException Exception, etc.**

---

```
try
{ // allocation code here
    public void DisplayNumbers() throws
    NumberFormatException{
        //display code here that uses number
        formatting }
```

```
        public int FindMaxArray( int a[] ,
int n) throws ArrayOutOfBoundsException{
        //FindMaxArray code here that uses
Array processing
    }
catch (NumberFormatException e1){ }
catch ( ArrayOutOfBoundsException e2) {
}
A try block can be nested just like for
loop.
```

---

## **Example 20.4: Nesting of Try Blocks**

```
try // outer try
{ // allocation code here
    try{ // inner try
        // inner try block code here
that uses File IO Statement}
        public void HandleFile()
throws FileNotFoundException{
            // file handling code here
that uses opening of file}
    } catch
```

```
(FileNotFoundException object) { }  
}  
catch (IOException object){ }
```

---

## 20.5 Using Finally Block

Usually when an exception occurs, the exception handling code may initiate rethrow to system or calling program. At that instant of time, there are several actions to be performed like closing of all open files, restoring the state to a state prior to occurring of the exception, etc. This is accomplished by using the `finally()`, as shown in the next example. It is included after the last catch block.

### **Example 20.5: Using of Finally Block**

```
try  
{ // allocation code here
```

```

        public void DisplayNumbers() throws
NumberFormatException{
            //display code here that uses number
formatting }
        public int FindMaxArray( int a[] ,
int n) throws ArrayOutOfBoundsException
exception{
            //FindMaxArray code here that uses
Array processing
        }
        catch (NumberFormatException e1){ }
        catch ( ArrayOutOfBoundsException e2) {
}
        finally { // finally block
/* includes all house keeping code like
saving current state ,closing of all
open
objects and file and restoring the state
to a state that existed prior to
occurring of the
problem*/
        } // end of finally block

```

---

## 20.6 Throw Exceptions

A programmer can explicitly throw an exception. Throw can be for any of the exceptions provided by `java.lang.Exception` package or user-

defined exception class. In the example that follows, we will show the user throwing an exception object for `ArithmeticException` class.

## Example

### 20.6: MultiCatchFinally.java A Program to Show Multiple Catch Statements and Finally Block

```
1. package com.oops.chap20;
2. import java.util.*;
3. public class MultiCatchFinally {
4. public static void main(String[]
args){
5. double mass,area,density=0.0;
6. boolean yesno=true;
7. try{ // create arrays for mass
,area,density
8. double [] massArray = new double[]
{200.00, 300.00,400.00,500.00};
9. double [] areaArray = new double
[] { 20.0,30.0,40.0,0.0};
10. double [] densityArray = new
double [massArray. length];
```



```

11.  int len = massArray.length;
12.  //for( int i=0; i<len+1;i++){
13.  for( int i=0; i<len;i++){
14.  if( areaArray[i]==0) throw new
ArithmeticException("Divide By Zero");
15.  else
16.
densityArray[i]=massArray[i]/areaArray[i
];}
17.  System.out.println("\n\t
mass\tarea\tdensity");
18.  for( int i=0; i<len;i++){
19.
System.out.println("\t"+massArray[i]+" \t
"+areaArray[i]+" \t"+densityArray[i]);}
20.  }catch(ArithmeticException e)
21.  {System.out.println("Divide By
Zero Exception has occurred ....");}
22.
catch(ArrayIndexOutOfBoundsException e)
23.  {System.out.println("Array out
of Bounds Exception has occurred ....");}
24.  finally{
25.  System.out.println("We are
inside finally block & exiting the
programme.....");}
26.  }// end of main
27.  }//end of class
Run 1 We have commented out Line No.
12 to test array

```

```

//index out of bounds exception

```

```

OUTPUT1 :Divide By Zero Exception has

```

occurred ....

We are inside finally block & exiting the programme.....

Run 2 We have commented out Line No. 13,14,and 15 to test Divide

//By Zero exception i.e  
ArithmeticException

OUTPUT2: Divide By Zero Exception has occurred ...

We are inside finally block & exiting the programme.....

<b>L i n e</b>	is try block and multiple catch blocks are at line Nos. 20 & 22.
----------------------------	--

<b>N o . 7 :</b>	
----------------------------------	--

<b>L i n e</b>	we have purposely defined areaArray[3] =0.0 to simulate the divide by zero Error.
----------------------------	---

**N  
o  
.  
9  
:**

**L** `for( int i=0; i<len+1;i++){` shows that  
**i** we are trying to go beyond the array size `i.e`  
**n** `len`. It will trigger  
**e** `ArrayIndexOutOfBoundsException`.

**N  
o  
.  
1  
2  
:**

**L** checks `if( areaArray[i]==0)` and if true  
**i** throw new `ArithmeticException`. The  
**n** exception object is caught by catch block at line  
**e** no. 20. This is user thrown exception object. The  
Exception class is provided by  
**N** `java.lang.Exception` package.

**N  
o  
.  
1  
4  
:**

## 20.7 Throws Exceptions

There is another variation called throws provided by java. We have been using this feature in all our IO related programs like `public static void main(String [] args) throws IOException{`. All checked exceptions, i.e., all checked exceptions are required to be handled by the user. However, if the user does not want to handle the exception, then he has to throw out using throws clause.

### Example

#### **20.7: MultiCatchFinally.java a Program to Show Multiple Catch Statements and Finally Block**

---

```
package com.oops.chap20;
import java.io.*;
import com.oops.chap20.OurOwnException;
import java.lang.Exception;
import java.util.*;
class CheckCredits extends
```

```

OurOwnException{
public void FindBal(String id,double
trAmt) throws OurOwnException{
    double[] BanBal=new double[]
{2000.0,5000.00,900.00,90000.0};
    String[] EmpIdNo=new String[]{
"50595","50596","50597","50598"};
    for(int i=0;i<EmpIdNo.length;i++){
        if( EmpIdNo[i].compareTo(id)==0)
        { System.out.println("Bank balance"
+BanBal[i]);System.out.
        println("Trnsaction Amount " +trAmt
);
            if ( BanBal[i]- trAmt <1000.00)
                throw new
OurOwnException("Sorry Transaction can
not be
        processed Bal<1000.00");
            else System.out.println("Successful");
        }
        else continue;
    }// end of for
} // end of FindBal
} // end of CheckCredits class
class MyException {
public static void main(String[]
args)throws IOException{
String idNo, amt; boolean yesno=true;
double transAmt;// variables for input
data
BufferedReader input = new
BufferedReader (new InputStreamReader

```

```

(System.in));
System.out.println("Enter id number of
Employee");
String stg =
input.readLine();stg.trim();
System.out.print("Enter Transaction
amount :");
amt=input.readLine();amt.trim();
transAmt=Double.parseDouble(amt);
// make an object of CheckCredits class
CheckCredits obj=new CheckCredits();
// call FindBal
try{
obj.FindBal(stg,transAmt);
}catch(OurOwnException e){
    System.out.println("Inside catch block
- Our Own Exception");
    System.out.println(e.getMessage());}
    }// end of main
}// end of class MyException
OUTPUT : Enter id number of Employee
50596
Enter Transaction amount :25000.00
Bank balance5000.0 Trnsaction Amount
25000.0
Inside catch block - Our Own Exception
Sorry Transaction cannot be processed
Bal<1000.00

```

---

## 20.8 Re-throwing of An Exception

Suppose an exception occurs at some method but the corrective mechanism is placed at some other method, say `main()` method, then the exception caught by catch block of called method can re-throw, so that `main()` will catch the re-thrown exception and carry out the correcting mechanism. Example 1 and 2 show how to catch re-thrown exceptions for String and Array.

## 20.9 Defining Our Own Exception

On many occasions, we need to define our own exception handlers in project development work like exception during data validations, data validations, etc. We need to throw exception object and then write a catch block to capture this exception. Exception class is super for all exceptions that occur. The class hierarchy for Exception classes is shown in Figure 20.1.

### *20.9.1 Procedure for Throwing Our Own Exceptions*

- **Step 1:** import Exception class : **import**  
java.lang.Exception;
- **Step 2:** Make your own class extend to Exception  
class: class OurOwnException **extends**  
Exception.
- **Step 3:** Define constructor for our class.  
OurOwnException(String stg){**super**(stg);}
- **Step 4:** Create a try block where exception  
likely to occur
- **Step 5:** If exception occurs throw our own  
exception by creating exception object:  
**throw new** OurOwnException("Sorry  
Transaction can not be processed. Bal is  
<1000.00");
- **Step 6:** Use catch block to capture the exception object  
thrown: **catch** (OurOwnException e)  
{System.out.println("Inside a catch  
block- OurOwnException");}
- **Step 7:** Get the exception message :  
System.out.println(e.getMessage());}

## Example 20.8: OwnException.java

### A Program to Show How to Throw Our Own Exception



---

```
1. package com.oops.chap20;
2. import java.lang.Exception;
3. class OurOwnException extends
Exception{
    4. OurOwnException(){} // default
constructor
    5. OurOwnException(String stg)
{super(stg);} // constructor with arg
    6. }// end of own exception
    7. class OwnException1{
    8. public static void main(String[]
args) {
    9. double credits = 20000.00;
   10. double debits = 19900.00;
   11. try{
   12. if ( ( credits-debits)<1000.00 )
   13. throw new OurOwnException("Sorry
Transaction can not be processed. Bal is
<1000.00");
   14. else
   15. System.out.println("Transaction
sucessful");
   16. }catch(OurOwnException e)
   17. {System.out.println("Inside a
catch block- OurOwnException");
   18.
System.out.println(e.getMessage());}
   19. }//end of main
   20. }// end of ownException1
    OUTPUT : Inside a catch block-
OurOwnException
```

Sorry Transaction can not be  
processed. Bal is <1000.00

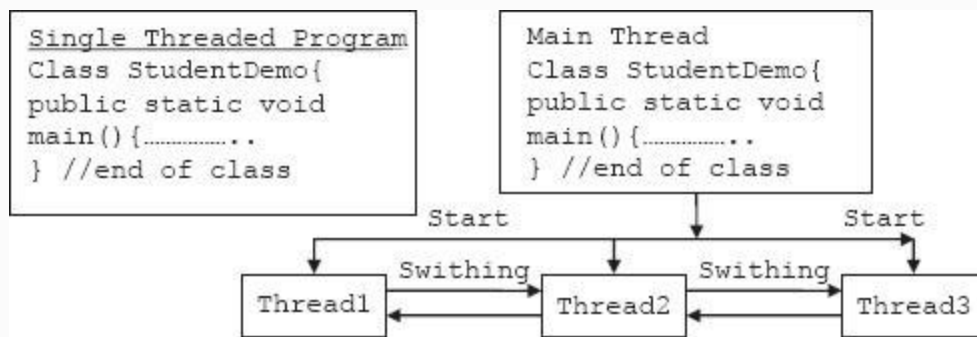
<b>Line No</b> <b>3:</b>	<code>class OurOwnException extends Exception{ shows OurOwn exception class extending to java.lang.Exception class.</code>
-----------------------------	--

## 20.10 Concepts of Multithreading

You are familiar with picture in picture facility provided by television wherein while the newsreader is presenting the news, a small window appears with a totally different news item or amplification of what the newsreader is presenting, probably cricketing action. How is this possible? Surely you need two independent executions.

What is a thread? Simply put, thread to Java is the same as process to an operating system. Thread is a sequence of instructions

that will be executed when the CPU is scheduled to handle the thread. The `void main()` programs so far we have been executing were indeed executed by Java on a single thread. This concept is shown in Figure 20.3.



**Figure 20.3** Multithreaded program

### *20.10.1 Multithreaded Program*

When more than one thread is executing independent processes, it is called a multithreaded program. Why is multithreaded programming required? Multithreading is useful in any situation where a programmer requires more than one task to be handled at the same time.

Situations like games programming, foreground and background jobs, concurrent handling of multiple outputs, etc., are ideally suited for multithreaded programming. Threaded and multithreaded programs are ideally suited for client server programs wherein the server handles clients residing on several threads.

## 20.11 Process vs Threads

### *20.11.1 Process*

A process is a job in hand. We can also say that this is a simple sequence of instructions to be executed by the CPU. Operating system creates several processes. CPU is then scheduled to handle different processes based on the priorities of the processes by OS. OS switches CPU amongst processes and allocates CPU time and completes all the tasks. When process shifting takes place, it is necessary to save the context of the current execution like stack, variables, status, etc. As context switching is involved, processes are called heavyweight processes.

## 20.11.2 Threads

Java brings concurrent processing, not parallel, just like OS does with its processes, within the preview of programmers by innovative and unique feature called multithreaded programming. In multithreaded programs, the main program is run on a main thread but the main program creates threads that run independent and concurrent program flows. It is the main program that creates and starts the other three threads, as shown in Figure 20.3. Once started, the three subthreads run independently as per priorities set and as per scheduling of CPU concurrently sharing CPU time and other common resources.

Threads are also called light weight because they consume less memory resources. Why because there is no context switching involved in multithreaded programming.

## 20.12 How to Create and Run the Threads?

By default, our main program is running on the main thread. To create our own threads, the following steps are used:

- **Step 1:** A thread can be created either by extending to Thread class or implementing the Runnable Interface.  
**Example:** class ThreadTest extends Thread or  
Class ThreadTest implements Runnable
  - **Step 2:** Inside the class define run () method. In side run method, we can implement thread functionality. Run () method is automatically recognized and executed by the thread.
- 

```
class ThreadTest extends
Thread{ or
class ThreadTest extends
implements Runnable{
public void run()
{.....}
```

---

Note that run method is where the entire action for the thread will take place. It is like the main () method. It can declare its own variables, instantiate other classes. Run is the entry point for any new thread that is created. The Thread created ends when the run () ends.

- **Step 3:** Create an object of ThreadTest in main(). This object is used to invoke the run method
- 

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

```
ThreadTest obj =new  
ThreadTest();
```

---

- **Step 4:** Create the thread in main and attach thread to the object.
- 

```
Thread thrd = new  
Thread(obj); OR  
Thread thrd = new Thread( obj  
, "threadname");
```

---

- **Step 5:** Run the thread by calling start method on the thread
- 

```
Thrd.start();
```

---

We can suspend the thread from execution for a specified period of time by using `sleep()` method. The syntax is:  
`static void sleep(milliseconds)`  
**throws** `InterruptedException` **or** `static void sleep (milliseconds, nanoseconds)` **throws** `InterruptedException`. As `sleep` is a static method, we can invoke directly as:  
`Thread.sleep(ms)` ; During the time

thread is suspended, CPU is allocated to other threads.

Thread class also provides `getName()` and `setName()` methods, for example, `Thread.getName()` and `Thread.setName("name")`. The methods by Thread class and Runnable interface are shown in Table 20.2.

**Table 20.2** Methods of thread class and runnable interface



Method	Remarks
--------	---------

<code>getName ();</code>  <code>setName (stg);</code>	Obtain and set names for the threads
<code>getPrio rity()</code>  <code>setPrio rity()</code>	Get & set methods to obtain and set Priorities
<code>isAlive ()</code>  <code>join()</code>	To check if thread is still running  It throws <code>InterruptedException</code> . Calling thread waits till specified thread joins it.
<code>sleep(m s)</code>  <code>sleep(m sec,nse c)</code>	Suspend execution of thread for ms duration Suspend execution of thread for ms & nano sec duration

We implement the creation of threads using extends method in Example 20.8. We show the implementation using Runnable interface in Ex 20.4. Runnable is an interface provided to create the threads. The programmer has to execute overridden `run()` to create and run the thread.

## Example

### 20.9: FirstThreaddemo.java A Program to Show How to Create and Run Thread. Use Extends Feature to Create Threads

```
1. package com.oops.chap20;
2. class FirstThread extends Thread{
3.     public void run(){
4.         // Thread executes the sequence
5.         try{
6.             for (int i=0;i<4;i++)
7.                 {System.out.println(" thread1 :" +
i + " : ");
8.                 Thread.sleep(600);
9.             }
```

```

10.} catch(InterruptedException e )
11. {System.out.println("thread1
interrupted");}
12. System.out.println("Exiting from
thread1");
13. }//end of run
14. }//end of FirstThread
15. class SecondThread extends
Thread{
16. public void run(){
17. try{
18. for (int i=0;i<4;i++)
19. { System.out.println(" thread2
:" + i + " : ");
20. Thread.sleep(600);
21. }
22. }catch(InterruptedException e )
23. {System.out.println("thread2
interrupted");}
24. System.out.println("Exiting from
thread2");
25. }//end of run
26. }//end of SecondThread
27. public class FirstThreadDemo {
28. public static void main(String[]
args) {
29. //create an object of
FirstThread & SecondThread
30. FirstThread obj1 = new
FirstThread();
31. SecondThread obj2 = new
SecondThread();

```

```

    32.  //create the Thread and attach
it to object
    33.  Thread thread1 = new Thread(
obj1);
    34.  Thread thread2 = new Thread(
obj2);
    35.  //Run the thread by calling
start() which calls run()
    36.  thread1.start();
    37.  thread2.start();
    38.  try{
    39.  for (int i=0;i<4;i++)
    40.  { System.out.println("
threadMain : " + i + " : ");
    41.  Thread.sleep(1200);
    42.  }
    43.  }catch(InterruptedException e )
    44.  {System.out.println("main thread
interrupted");}
    45.  System.out.println("Exiting from
main thread");
    46.  }//end of main
    47.  }//end of FirstThreadDemo
Output: threadMain :0 :
        thread1 :0 : thread2 :0 : thread2 :1
: thread1 :1 : threadMain :1
        thread2 :2 : thread1 :2 : thread1 :3
: thread2 :3 : th readMain :2
    Exiting from thread1 Exiting from
thread2 threadMain :3 :
    Exiting from main thread

```

---

<b>Li ne N os</b> <b>· 30 &amp; 31 :</b>	instantiate FirstThread and SecondThread. These threads extend to Thread class.
<b>Li ne N os</b> <b>· 33 &amp; 34 :</b>	create the Thread objects and attach them with objects: Thread thread1 = new Thread( obj1);
<b>Li ne N os</b> <b>· 36 &amp; 37 :</b>	starts the thread which in turn call the run() method. Both thread1 and thread2 start running.

<b>Line Nos</b> . <b>38</b> – <b>45</b> :	describe main thread. Note that at line no: 41 <code>sleep(1200)</code> ensures that main thread sleeps for 1200 seconds. During this time other threads are handled by CPU.
<b>Line Nos</b> <b>6:</b>	<code>FirstThread</code> in side <code>run()</code> methods executes a for loop as part of its programme. After each loop, the thread goes to sleep for 600 ms to facilitate <code>SecondThread</code> and <code>MainThread</code> to get their share of CPU.
<b>Line Nos</b> . <b>15</b> – <b>26</b> :	declare a class <code>SecondThread</code> identical to <code>FirstThread</code> .

Note that both threads wait only for 600 ms and main thread waits for 1200ms . his

timings are deliberately chosen to ensure that main thread finishes last.

### *20.12.1 Which is Better: Extends Thread or Implements Runnable?*

Use Inheritance extends Thread method if you have anything new to add in subclass. Else implementing Runnable is a better option. In our next example, we will introduce constructors to create threads. We will use Runnable interface:

#### **Example**

**20.10: SecondThreadDemo.java A Program to Show How to Create and Run Thread. Use Extends Feature to Create Threads**

---

```
1. package com.oops.chap20;
2. class FirstThread2 implements
Runnable{
3. Thread thrd1;
4. FirstThread2() {
```

```

5. System.out.println("Creating
FirstThread2");
6. thrd1 = new
Thread(this, "FirstThread2");
7. System.out.println("Creating
FirstThread2"+ thrd1);
8. thrd1.start();
9. }
10. public void run(){
11.    // FirstThread2 executes the
sequence
12.    try{
13.        for (int i=0;i<4;i++)
14.        {System.out.println(" thread1 : "
+ i + " : ");
15.        Thread.sleep(600);}
16.    }catch(InterruptedException e )
17.    {System.out.println("thread1
interrupted");}
18.    System.out.println("Exiting from
thread1");
19.    }//end of run
20. }//end of FirstThread
21. class SecondThread2 implements
Runnable{
22.    Thread thrd2;
23.    SecondThread2(){
24.        System.out.println("Creating
SecondThread2");
25.        thrd2 = new
Thread(this, "FirstThread2");
26.        System.out.println("Creating

```



```

SecondThread2"+ thrd2);
    27.  thrd2.start();
    28.  }
    29.  public void run(){
    30.  System.out.println("thread1
Details");
    31.  System.out.println("Name
:"+thrd2.getName());
    32.  System.out.println("Priority
:"+thrd2.getPriority());
    33.  try{
    34.  for (int i=0;i<4;i++)
    35.  { System.out.println(" thread2
:" + i + " : ");
    36.  Thread.sleep(600);
    37.  }
    38.  }catch(InterruptedException e )
    39.  {System.out.println("thread2
interrupted");}
    40.  System.out.println("Exiting from
thread2");
    41.  }//end of run
    42.  }//end of SecondThread
    43.  public class ThreadsConstDemo {
    44.  public static void main(String[]
args) {
    45.  //create an object of
FirstThread2 & SecondThread2
    46.  new FirstThread2();
    47.  new SecondThread2();
    48.  try{
    49.  for (int i=0;i<4;i++)

```

```

50.  { System.out.println("
threadMain : " + i + " : ");
51.  Thread.sleep(1200);
52.  }
53.  }catch(InterruptedException e )
54.  {System.out.println("main thread
interrupted");}
55.  System.out.println("Exiting from
main thread");
56.  }//end of main
57.  }// end of ThreadRunnableDemo
Output: Creating FirstThread2
Creating
FirstThread2Thread[FirstThread2,5,main]
thread1 :0 :
Creating SecondThread2
Creating
SecondThread2Thread[FirstThread2,5,main]
threadMain :0 :
thread1 Details
Name :FirstThread2
Priority :5
thread2 :0 : thread1 :1 : thread2 :1
: thread1 :2 :
threadMain :1 : thread2 :2 : thread1
:3 : thread2 :3 :
Exiting from thread1 threadMain :2 :
Exiting from thread2
threadMain :3 : Exiting from main
thread

```

---



---

<b>Line Nos. 46 &amp; 47:</b>	creates new thread by : new FirstThread2 (); new Second Thread2 ();
<b>Line No. 6:</b>	thrd1 = new Thread(this, "FirstThread2"); attaches new thread created with this (current thread) and names it as FirstThread2
<b>Line No. 8:</b>	starts the thread1. Yhread enter run () method.
<b>Line Nos. 30–32:</b>	gets you thread details

```

System.out.println("thread1 Details");
System.out.println("Name
:"+thrd2.getName()); System.out.println
("Priority :"+thrd2.getPriority());

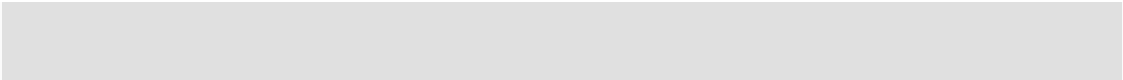
```

## 20.12.2 Use of *isAlive()* and *join()* Methods

The `sleep()` method introduced suspends a thread for duration specified and CPU is shifted to other waiting threads. The selection of duration is arbitrary and CPU is forcibly pulled out from thread after time duration. This is not a satisfactory solution. Java provides: `final boolean isAlive()` method to test if the thread is alive. Java also provides a more elegant method called `final void join()` throws `InterruptedException`. This is a non-greedy method and it waits till the thread on which it is called is terminated. Here the calling thread waits till the specified thread joins it.

### **Example**

**20.11: `IsAliveJoinDemo.java` A Program to Show How to Create and Run Thread. Use Extends Feature to Create Threads**



---

```
1. package com.oops.chap20;
2. import
com.oops.chap20.FirstThread2;
3. import
com.oops.chap20.SecondThread2;
4. public class IsAliveJoinDemo {
5. public static void main(String[]
args) {
6. FirstThread2 obj1= new
FirstThread2();
7. SecondThread2 obj2 =new
SecondThread2();
8. // check if threads are alive.
thrd1&2 is objects of
First/SecondThread2
9.
System.out.println(obj1.thrd1.isAlive())
; //thrd1 is object of
FirstThread2
10.
System.out.println(obj2.thrd2.isAlive())
;
11. // wait for threads to finish.
Use join method
12. try{
13. System.out.println("\nwaiting
for threads to finish");
14. obj1.thrd1.join();
```

```

15.  obj2.thrd2.join();
16.  }catch(InterruptedException e )
17.  {System.out.println("main thread
interrupted");}
18.  System.out.println("Exiting from
main thread");
19.  }//end of main
20. }// end of ThreadRunnableDemo
Output: Creating FirstThread2
Creating
FirstThread2Thread[FirstThread2,5,main]
thread1 :0 : Creating SecondThread2
Creating
SecondThread2Thread[FirstThread2,5,main]
True true
thread1 Details waiting for threads
to finish
Name :FirstThread2 Priority :5
thread2 :0 : thread1 :1 : thread2 :1
: thread1 :2 : thread2 :2 :
thread1 :3 : thread2 :3 : Exiting
from thread1Exiting from thread2
Exiting from main thread

```

---

## 20.13 Life Cycle of Thread

There are five states for thread to be in. They are: NewBorn), Runnable, Running, Blocked, and Dead State. **A running**

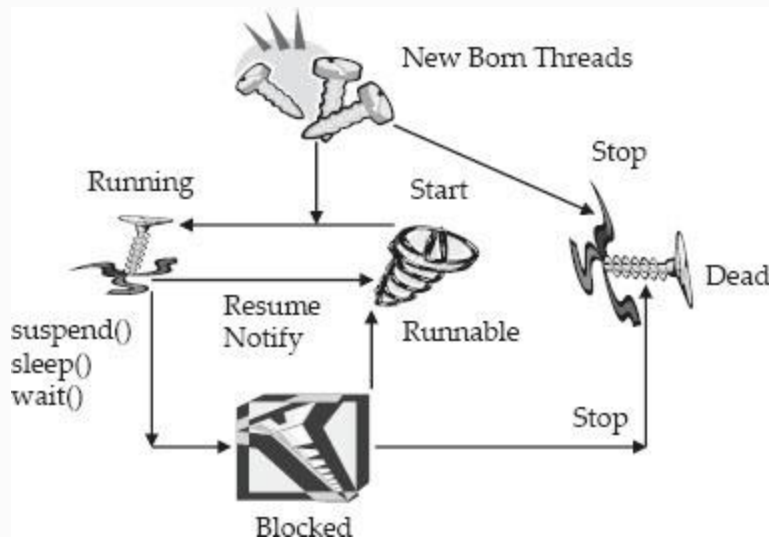
thread can be stopped by `stop()` method. A thread can be blocked by any one of the following methods:

- `sleep()`
- `suspend()` . //deprecated . Cannot be used due to inherent problems. Instead Java recommends using of `IsAlive()` and `join()` methods.
- `wait()` ; A thread is blocked until certain conditions prevail

A newborn thread goes to either Runnable state or Dead State. With `start()` method the thread goes to Runnable state. Runnable state means that the thread is ready and waiting in the queue to be serviced by CPU. Depending on the priority or on the basis of round robin the threads are scheduled to be serviced by CPU. The thread then goes to the running state.

Now `sleep()` or `suspended()` or `wait()` takes the thread to a blocked state. Blocked state is one which will not be handled by a CPU. Resume or `notify()` or `notifyall()` will take the thread to runnable state. `Notify()` and

`notifyall()` are methods invoked by thread that has completed its job with CPU and issues `notify()` or `notifyall()`. Depending on the priority, the thread with the highest priority gets CPU's attention.



**Figure 20.4** The states that a newborn thread goes through

- `thrd.notify()` : releases the object and informs the thread waiting in a queue that object is available.
- `Thrd.notifyAll()`: Releases the object and sends notification to all waiting threads
- `Obj.wait()` : Thread waits till it receives `notify()` or `notifyAll()` methods.



## 20.14 Thread Priorities

Thread priority decides the order of scheduling of thread waiting. Thread priority is from No. 1 to 10. We can set the priority with:

```
ThreadName.setPriority(int );  
or ThreadName.setPriority(NORM  
_PRIORITY );
```

Int can be any number between 1 and 10.

---

```
MIN_PRIORITY =1 : NORM_PRIORITY =5  
MAX_PRIORITY =10
```

---

Java executes the highest priority thread first. The lower priority thread gets CPU attention when higher priority thread is stopped or `sleep()` is used or `wait()` is executed. When a thread is running if higher priority thread arrives, it preempts the running thread and schedules the higher priority thread. The running thread goes to Runnable state.

## Example 20.12: Priority.java: A program to Show How to Set Priorities to the Threads

```
1. package com.oops.chap20;
2. class MaxPriority implements
Runnable{
3. Thread thrd1;
4. MaxPriority(){
5. System.out.println("Creating
MaxPriority");
6. thrd1 = new
Thread(this, "MaxPriority");
7.
thrd1.setPriority(Thread.MAX_PRIORITY);
8. System.out.println("Creating
MaxPriority"+ thrd1);
9. thrd1.start();
10. }
11. public void run(){
12. System.out.println("Maximum
Thread Details");
13. System.out.println("Name
:"+thrd1.getName());
14. System.out.println("Priority
:"+thrd1.getPriority());
```

```

15.  // MaxPriority executes the
sequence
16.  try{
17.    for (int i=0;i<4;i++)
18.    {System.out.println(" thread1 : "
+ i + " : ");
19.    Thread.sleep(600);}
20.  }catch(InterruptedException e )
21.  {System.out.println("thread1
interrupted");}
22.  System.out.println("Exiting from
thread1");
23.  }//end of run
24.  }//end of FirstThread
25.  class MinPriority implements
Runnable{
26.    Thread thrd2;
27.    MinPriority(){
28.      System.out.println("Creating
MinPriority");
29.      thrd2 = new
Threadthis, "FirstThread2");
30.
thrd2.setPriority(Thread.MIN_PRIORITY+2)
;
31.      System.out.println("Creating
MinPriority"+ thrd2);
32.      thrd2.start();
33.    }
34.    public void run(){
35.      System.out.println("minimum
thread Details");

```

```

36.  System.out.println("Name
:"+thrd2.getName());
37.  System.out.println("Priority
:"+thrd2.getPriority());
38.  try{
39.    for (int i=0;i<4;i++)
40.    { System.out.println(" thread2
:" + i + " : ");
41.    Thread.sleep(600);
42.    }
43.  }catch(InterruptedException e )
44.  {System.out.println("thread2
interrupted");}
45.  System.out.println("Exiting from
thread2");
46.  }//end of run
47.  }//end of SecondThread
48.  public class Priority {
49.    public static void main(String[]
args) {
50.    //create an object ofMaxPriority
& MinPriority
51.    new MaxPriority();
52.    new MinPriority();
53.    try{
54.    for (int i=0;i<4;i++)
55.    { System.out.println("
threadMain :" + i + " : ");
56.    Thread.sleep(600);
57.    }
58.  }catch(InterruptedException e )
59.  {System.out.println("main thread

```

```

interrupted");}
    60.  System.out.println("Exiting from
main thread");
    61.  }//end of main
    62.  }// end of Priority.java
Output: Creating MaxPriority
Creating
MaxPriorityThread[MaxPriority,10,main]
Maximum Thread Details
Name :MaxPriority
Priority :10
thread1 :0 :
Creating MinPriority
Creating
MinPriorityThread[FirstThread2,3,main]
threadMain :0 :
minimum thread Details
Name :FirstThread2 Priority :3
thread2 :0 : thread1 :1 : thread2 :1
: thread1 :2 : threadMain :1 : thread2
:2 : thread1 :3 : thread2 :3 : Exiting
from th read1
threadMain :2 : Exiting from thread2
threadMain :3 :
Exiting from main thread

```

<b>Line Nos. 51 &amp;</b>	<b>create two threads called MacPriority and MinPriority.</b>
-------------------------------	---

<b>52:</b>	
<b>Line No. 7:</b>	Constructor allocates priority as <code>MAX_PRIORITY</code> i.e. 10 to <code>thr1</code>
<b>Line No. 30:</b>	Constructor allocates priority as <code>MIN_PRIORITY+2</code> , i.e., 3 to <code>thread 2</code>
<b>Line Nos. 12 to 14:</b>	provide the details of <code>thread1</code> such as name, priority, etc.

As per priorities set, if you observe the output `thread1` with priority of 10 is completed First. The main thread by default has `NORM_PRIORITY = 5` which is higher than `Thread2` priority which is `MIN_PRIORITY + 2`, i.e., 3. Hence the main thread completes before `thread2`.

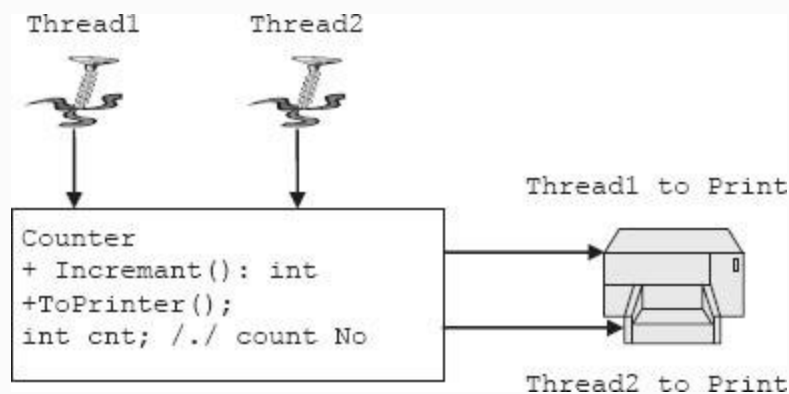
## 20.15 Synchronization

When multiple threads act on different objects, we would call it multitasking. As per

priorities allocated and as per CPU scheduling, the threads execute different objects and produce results. There is no problem in this case.

Suppose multiple threads act on the same object. This could lead to problems. Let us say that there are two threads, `thread1` & `thread2`, acting on the object, i.e., they will concurrently execute the same `run()` command. Each thread has only one job, that of incrementing the counter and sending it to the printer for printing. The output we are expecting is: `thread1 1001, thread2: 1001; thread:1002, thread2: 1002` and so on. Remember thread is on CPU and speed of execution is of the order of nanoseconds and the printer is a slow device. Let us say that counter has an initial value of 1000. After incrementing by `thread1` to 1001 and before the count has been printed, i.e., 1001, the control may pass to `thread2` due to `sleep()` or `wait()`, etc., which will increment the counter to 1002. The printer will look for count and print 1002 instead of 1001 which we

normally expect, thus leading to erroneous results. A second scenario is that after printing 1001, if the CPU is still with thread1, it will increment once again and print 1002, thus leading again to erroneous results. Multiple threads working independently on a single object and plausible problems that can arise are shown in Ex 20.6.



**Figure 20.5** Multiple objects on a single object

Java has provided a feature called synchronization that helps us to solve the problems of multiple threads sharing common resources and using the same object like printer. For example, we could



```
use: synchronized void  
UpdateCount() { // code to be  
synchronized here... }
```

Java creates a monitor for the `UpdateCount()` and hands its over to thread that has entered the object. It is like a key. The other objects join a queue waiting for their turn to enter the Member method `UpdateCount()`. Thus there is no conflict. But what is the price of synchronization? Simply the process has become serial handling of the thread rather than concurrent handling. It is similar to Railway trains waiting at the platform to cross a bridge that has only one track. A second train cannot enter the bridge section until and unless the train completes the bridge section and notifies signal) it to that effect. operations.

## **Example**

**20.13: SynchronisedDemo.java: A Program to Show Use of**

# Synchronized Block for Solving Problems of Multiple Threads Acting on Single Object

```
1. package com.oops.chap20;
2. class Sync3 extends Thread{
3. Thread thrd;
4. Sync3(String stg){
5. super(stg);}
6. public void run(){
7. synchronized(this) {
8. try{
9. for(int i=1000;i<1010;i++)
10.
11. {System.out.println(this.getName() + (i)
12. + " : ");
13. Thread.sleep(500);
14. }
15. }catch(InterruptedException e )
16. {System.out.println("thread1
17. interrupted");}
18. System.out.println("Exiting from
19. thread1"+this.getName());
20. }//end of try
21. }//end of run
22. }
```

```

args) {
    21.  //create an object ofMaxPriority
    & MinPriority
    22.  Sync3 obj1 =new
Sync3("Sync1");obj1.start();
    23.  Sync3 obj2 = new
Sync3("Sync2");obj2.start();
    24.  }//end of main
    25.  }// end of class
SynchronisedDemo
    Sync11000 : Sync20000 : Sync11001 :
Sync20001 : Sync11002 :
    Sync20002 : Sync11003 : Sync20003 :
Sync11004 : Sync20004 :
    Sync11005 : Sync20005 : Sync11006 :
Sync20006 : Sync11007 :
    Sync20007 : Sync11008 : Sync20008 :
Sync11009 : Sync20009 : Exiting from
thread1Sync1 Exiting from thread1Sync2

```

<b>L i n e N o .</b>	Declares the block as <code>synchronized(this)</code> meaning it is synchronizing the object so that object holds the monitor and till it relinquishes the CPU other threads can not enter this block of code.
--	--

## 20.16 Inter-thread Communications

Threads need to communicate with one another either to pass arguments, synchronize and optimize the performance. There is definitely a speed mismatch between producer and consumer.

Consider for example the action of inputting data from the keyboard into the computer. The steps involved in this process include:

- Producer, i.e., the keyboard produces at a rate and speed the data is keyed in by the user. This is a very slow process because input from the user is involved. The producer fills up buffer and on completion notifies that it has finished filling up of the buffer.
- Computer is consumer since it picks up the data entered by the keyboard into the system. This is a very high-speed device. Hence, we need a buffer and that needs to be in a synchronized block. This buffer is the same as that being used by producer. We will use `StringBuffer` class object to work as buffer for our application. Refer to the example provided below.

## Example

### 20.14: `ProducerConsumer.java`: A Program to Show the Inter-thread Communications

```
1. package com.oops.chap20;
2. class Producer extends Thread{
3. // producer produces data and add
it to buffer based on the signal
4. StringBuffer buffer;
5. Producer(){ // constructor
6. buffer = new StringBuffer();}
7. public void run(){
8. // synchronize the buffer
9. synchronized(buffer)
10. {
11. //produce items
12. for( int i=1;i<=8;i++)
13. { try {
14. buffer.append("Item"+i + ",");
15. Thread.sleep(200);
16. System.out.println("Buffer
Position :"+ i + " Filled.");
17. }catch(Exception e){}
18. }//end of for
19. buffer.notify();
```

```

20.  }// end of synchronized block
21.  }///// end of run
22.  }// end of Producer
23.  class Consumer extends Thread{
24.  // Create a reference of
Producer class
25.  Producer prod;
26.  //Constructor
27.  Consumer(Producer prod)
28.  { this.prod=prod;}
29.  public void run(){
30.  synchronized(prod.buffer)
31.  {// wait for notification from
Producer
32.  try{
33.  prod.buffer.wait();
34.  }catch(Exception e){}
35.  System.out.println("Displaying
items from Buffer..");
36.  System.out.println(prod.buffer);
37.  }//end of sync
38.  }// end of run
39.  }//end of class Consumer
40.  public class
ProducerConsumerDemo extends Thread{
41.  public static void main(String[]
args) throws Exception {
42.  // create objects of producer
and consumer
43.  Producer prod = new Producer();
44.  Consumer cons = new
Consumer(prod);

```

```

45.  // create threads for Producer
and Consumer
46.  Thread thrdProd = new
Thread(prod);
47.  Thread thrdCons = new
Thread(cons);
48.  thrdCons.start();
49.  thrdProd.start();
50.  }// end of main()
51.  }// end of class

```

ProducerConsumerDemo

Output:

Buffer Position :1 Filled. Buffer  
Position :2 Filled.

Buffer Position :3 Filled. Buffer  
Position :4 Filled.

Buffer Position :5 Filled. Buffer  
Position :6 Filled.

Buffer Position :7 Filled. Buffer  
Position :8 Filled.

Displaying items from Buffer.

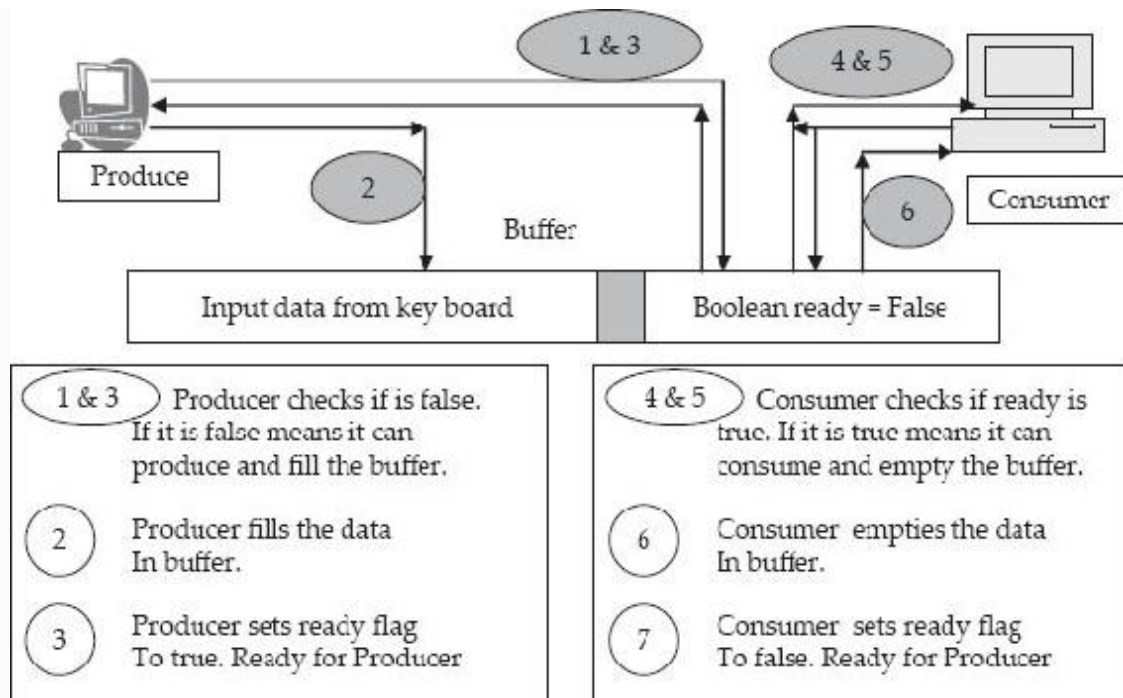
Item1,Item2,Item3,Item4,Item5,Item6,Item  
7,Item8

**Line  
Nos.**

create objects for Producer and  
Consumer.

<b>43 &amp; 44:</b>	
<b>Line Nos. 46 &amp; 47:</b>	create new thread objects <code>thrdProd</code> and <code>thrdCons</code> .
<b>Line Nos. 48 &amp; 49:</b>	start the threads.
<b>Line No. 4:</b>	creates <code>StringBuffer</code> object called <code>buffer</code> and constructor at Line No. 6 allocates resources.





**Figure 20.6** Producer consumer problem

Producer produces in a for loop continuously and fills up the buffer through append statement at Line No. 14. Note that buffer is in a synchronized block defined at Line No. 9.

Producer notify() at Line No. 19 to the threads waiting on the object.

Consumer creates an instance of Producer at Line No. 25 and constructor receives the

reference to Producer as argument and allocates resource at Line Nos. 27 and 28.

<b>Line No. 30 :</b>	block is synchronized with object of Producer buffer .
<b>Line No. 33 :</b>	consumer thread is waiting for notify() from the producer at Line No. 19. This is achieved by <code>wait()</code> command at Line No. 33 : <code>prod.buffer.wait();</code>

## 20.17 Deadlock in Multithreaded Programming

A deadlock will occur if there are two synchronized objects and if two threads working independently on these synchronized objects have circular dependency. Suppose there are two

synchronized objects, i.e., each object has a monitor. Let the objects be `Obj1` and `Obj2`. Let there be two threads, `thread1` and `thread2`, created which enter the respective monitors on `Obj1` and `Obj2`. Now `Obj1` calls any synchronized method on `Obj2`, the `thread1` will be blocked because `thread2` has the monitor. So far so good.

Now what happens if `thread2` calls a method on `Obj1`? Surely it will be blocked because `thread1` has the monitor on `Obj1`.

Thus, we can see a circular dependency on a pair of synchronized objects. A deadlock occurs. For a deadlock to occur, the following conditions are a must:

- Two synchronized objects.
- Two or more threads with monitor on the objects and a circular dependency.

A deadlock cannot be prevented but its occurrence can be minimized. When a deadlock occurs, the threads are immediately stopped, resources are released and processes are restarted.

## 20.18 Summary

1. Errors are of three types, viz., syntactical errors, logical errors, and bugs. Syntactical errors can be caught by compiler. Bugs can be fixed by assert statements and debugging.
2. Exceptions on the other hand are unusual conditions that occur at run time and can lead to errors that can crash a program.
3. Exceptions can be synchronous exceptions like `IOException`. These can be predicted.
4. Exceptions can be asynchronous exceptions like `OutOfMemoryException` that cannot be predicted.
5. Checked Exception are those that can be handled by compiler and need to be handled by the user. In case it is not handled, exception is to be thrown using throws clause.
6. A try block creates exception object and catch block catches the exception. Try block can be nested. A try block can initiate several exceptions.
7. A catch block can capture several exceptions.
8. Finally, block is placed after all catch blocks and can be used to do all housekeeping tasks such as closing all open files and saving status and displaying common essential messages before leaving try & catch blocks.
9. Throw exceptions: A programmer can explicitly throw an exception. Throw can be for any of the Exceptions provided by `java.lang.Exception` package or user-defined exception class.
10. Thread is a sequence of instructions that will be executed when CPU is scheduled for handling of the thread. Simply put, thread to Java is the same as process to an operating system.

11. Threads can be created using Runnable interface or by extending to Thread class. Runnable is used when nothing new is planned.
12. When more than one thread is executing independent processes, it is called multithreaded programs.
13. Process: A process is a job in hand. As context switching is involved, processes are called heavyweight processes.
14. Java provides: final boolean isAlive() method to test if the thread is alive.
15. Method called final void join() throws InterruptedException. This is a non-greedy method and it waits till the thread on which it is called is terminated. Here the calling thread waits till the specified thread joins it.
16. There are five states for the thread to be in. They are: NewBorn), Runnable, Running, Blocked, and Dead State.
17. A thread can be blocked by any one of the following methods: sleep(), suspend(), and block().
18. thrd.notify(): releases the object and informs the thread waiting in a queue that object is available.
19. Thrd.notifyAll(): Releases the object and sends notification to all waiting threads
20. Obj.wait(): Thread waits till it receives notify() or notifyAll() methods.
21. Thread priority decides the order of scheduling of thread waiting. MIN\_PRIORITY =1 : NORM\_PRIORITY =5  
MAX\_PRIORITY =10.
22. Java has provided a feature called synchronization that helps us to solve the problems of multiple threads sharing common resource and using the same object like printer.
23. Inter-thread communication is provided with the help of synchronization. A better technique is by using wait()

and `notify()` commands. This feature is useful for passing arguments between the threads.

## Exercise Questions

### Objective Questions

1. Which of the following statements are true in respect of Errors and Exceptions of Java?

1. Exceptions occur at compile time
2. Logical errors can be detected by compiler
3. Syntactical errors can be detected by compiler
4. Bugs can be detected by compiler

1. i, ii and iii
2. iii
3. ii and iii
4. ii and iii

2. Which of the following statements are true in respect of Exceptions of Java?

1. Synchronous exception cannot be predicted
2. Asynchronous exceptions cannot be predicted
3. IOException is synchronous exception
4. ArrayIndexOutOfBoundsException is asynchronous exception

1. i, ii and iii
2. iii
3. ii and iii
4. ii, iii and iv

3. Which of the following statements are true in respect of finally block of Java?

1. Finally closes all open objects
2. Finally block can be used to close all open files
3. Placed after try block
4. Placed after all catch blocks

1. i and iii

- 2. iii
- 3. ii and iv
- 4. ii, iii and iv

4. Try block can be nested

TRUE/FALSE

5. Catch block can be nested

TRUE/FALSE

6. All checked exceptions are required to be handled by user Or throw out the exception using throws clause

TRUE/FALSE

7. Which of the following statements are true in respect of Threads of Java?

- 1. Process is job in hand for OS
- 2. Thread to Java is what process is to OS
- 3. Thread is a heavyweight process
- 4. Shift of CPU from a thread involves saving of context

- 1. i and ii
- 2. iii
- 3. ii and iv
- 4. ii, iii and iv

8. Which of the following statements are true in respect of Threads commands of Java?

- 1. `IsAlive()` checks if thread is still running
- 2. `join()` throws `InterruptedException`
- 3. when `join()` is used calling thread waits till specified thread joins it
- 4. `join()` is a static method

- 1. i, ii and iii
- 2. i, iii and iv
- 3. ii and iv
- 4. ii, iii and iv

9. Which of the following statements are true in respect to synchronization of Java?

1. Block of statements can be synchronized and methods cannot be synchronized
  2. Erroneous results may occur when there is a speed mismatch of resources of threads
  3. Only one thread can enter a synchronized block
  4. Java creates and handover a monitor to thread entering synchronized block
- 
1. i, ii and iii
  2. i, iii and iv
  3. i and iii
  4. ii, iii and iv

### **Short-answer Questions**

10. What are errors?
11. What are exceptions?
12. How can bugs be removed?
13. Distinguish synchronous and asynchronous exceptions.
14. Distinguish checked and unchecked exceptions.
15. Explain throws and throw clause of exception mechanism.
16. Explain finally block usage.
17. Distinguish thread and Process.
18. Why is thread called a lightweight process?
19. In how many ways can a thread be created?
20. What is the entry point method for Thread execution?
21. Distinguish `suspend()` and `Stop()` methods.
22. `join()` and `wait()` are better than `suspend()` and `sleep()`. Why?
23. When do we use `extends Thread` method and implements `Runnable` interface?

### **Long-answer Questions**

24. Explain the working of java Errors and Exception detecting and correcting mechanism.
25. Explain uses of Try block, catch block and finally block with suitable examples.
26. Explain how you can deploy your own exception class.



27. How can Thread be created and run using extend clause and implementing runnable interface?
28. Explain the complete life cycle of thread.
29. What is synchronization? What is the price one pays if one uses synchronization?
30. What is Inter-thread communications? Why is this feature required and how is it achieved?
31. Explain the methodology of using `join()`, `wait()`, `notify()` and `notifyall()` to solve the problems associated with multithreaded programming.

#### **Assignment Questions**

32. Define an Exception called `wrongDateException` that is thrown when the date entered by the user is within permissible range such as dd 1 to 31, mm 1 to 12 and yy  $\geq 2000$  and  $\leq 2010$ .
33. Develop a `String` class that incorporates an exception `StringNotSame` that is triggered when a string is NOT equal to inputted `String`.
34. Develop `Stack` class with Exception thrown for `StackFullException` and `StackEmptyException`.
35. Explain the conditions in which a thread goes to a blocked state. Explain how a thread can return to `Runnable` state.
36. Rewrite Producer and Consumer Problem discussed with the help of a buffer and synchronization of block. Use `sleep()` or `wait()` / `notify()` to achieve the result.
37. Develop an application to show multithreaded program to show hours, minutes and seconds on separate threads.

#### **Solutions to Objective Questions**

1. b
2. d
3. c
4. True
5. False
6. True
7. a
8. a
9. d

# 21

## Java IO Files

### LEARNING OBJECTIVES

*At the end of the chapter, you will be able to understand and program*

- Concepts of Streams and Stream classes.
- Use `InputStream` and `OutputStream`.
- Random file handling.
- File IO and Errors and Exception in File IO.
- Handling of primitive data types with `DataInputStream` & `DataOutputStream`.
- Read objects from and write objects on to a file.
- Object Streams and Piped Streams.

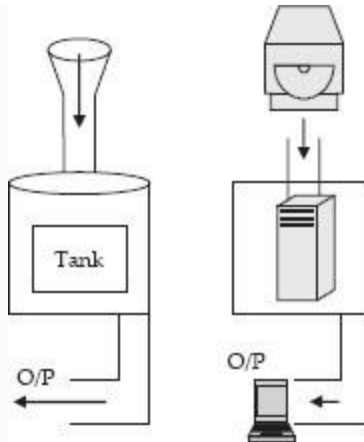
### 21.1 Introduction

We have all heard about streams of water; we call them springs flowing from top of hills to plains. In Java, flow of data from output device to computer system and computer to any one of the output devices is called Stream or IO streams.

This chapter will introduce you to concepts of `IOStreaming` and stream classes like `ByteStream` and `Character Stream` classes. We will introduce you to the concept of Random File IO. This chapter also discusses in detail File IO and Errors and Exceptions that are encountered while dealing with file. The topics on file handling presented includes input and output of characters, Bytes, primitive data types and Data Streams, and object Streams.

## 21.2 IO Streaming

In Java, IO streams are flow of bytes from one input device to Memory and vice versa. A stream is like a pipe. It can carry anything in the pipe, be it primitive data, objects, etc. Refer to Figure 21.1.

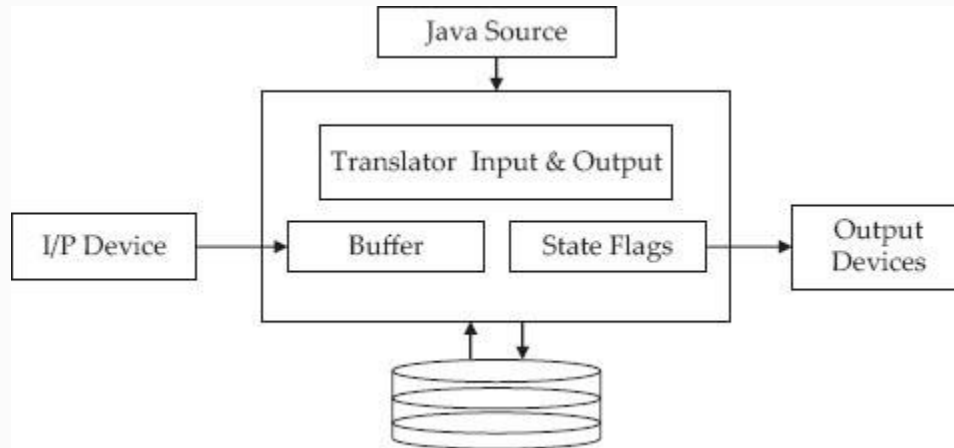


**Figure 21.1** Streams – a concept

The diagram on the left-hand part shows a water tank with input pipe and an output pipe. The diagram on the right-hand part shows input stream from CD drive and output stream to monitor.

Figure 21.1 takes a closer look at IO. For example, Java source is required to be converted by JRE to suit Input devices such as computer system. Hence input from IO devices are buffered first, then picked up by the computer system. Similarly, output from computer is buffered first and then picked up by slow-speed devices like printer,

memory, etc. Figure 21.2 provides a more detailed view of IO-streaming.



**Figure 21.2** IO Streaming – a close look

As an example, consider the case of inputting from keyboard. To read the data from the keyboard, we need to attach `InputStreamReader` to accept the input data from keyboard and to place it on buffer we need `BufferedReader`.

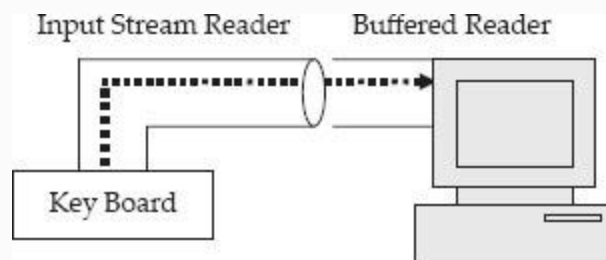
`DataInputStreamReader` accepts the data from keyboard and `System.in` represents keyboard. Therefore, we can write

---

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

---

**Figure 21.3** shows `IO StreamReader` for input from keyboard. `System` has three fields representing three different devices. They are:



**Figure 21.3** Input from keyboard

- `System.in`: Standard input is from keyboard. This represents `InputStream` Object.
- `System.out`: Console is represented by `System.out`. This represents `PrintStream` object.
- `System.err`: Java handles stream objects for input and output and hence is likely to encounter errors or exceptions. `System.err` is used to log errors that occur while handling streams.

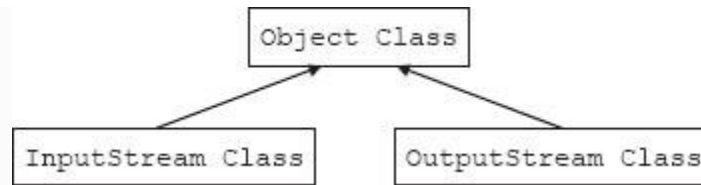
## 21.3 Java IO Stream Classes

Before we go into details of IO Streaming, it is necessary to review files. We would come across files everywhere we go. For example, colleges keep a file for each of their students. Similarly municipality holds files containing details of taxes to be paid by citizens. Indeed files are so common in our lives; Java language and other languages support files.

**What is a file?** A file is a collection of records.

`Java.io` package contains stream classes for handling File IO. Stream Files can be divided into two types: `ByteStream` classes and Text Files, also known as `CharacterStream` classes. These two streams are derived from Java's `Object` class, as shown in [Figure 21.4](#). [Table 21.1](#) gives details of classes. These Stream classes read from source and write to destination. The source and destinations are: Memory, pipe, and File. Pipes are used for handling threads in Java.





**Figure 21.4** Hierarchy of input and output stream classes

**Table 21.1** Java stream classes

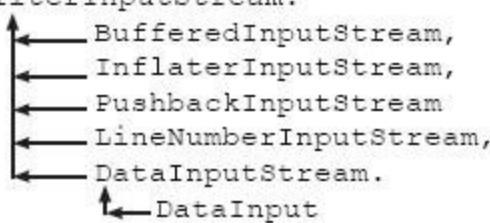
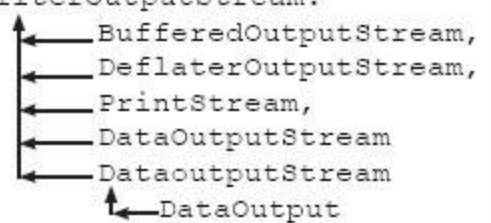
ByteStream Classes	CharacterStream Classes Text Files
FileInputStream	FileReader
FileOutputStream	FileWriter
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter

### 21.3.1 Classification of *ByteStream* Classes

The data is represented as byte ( 8 bits). All text files, and audio and video files that need to be transmitted using Internet, are stored

using ByteStream Classes. Table 21.2 provides details of input stream and output stream classes. Table 21.3 provides details of member methods Java InputStream/OutputStream classes.

**Table 21.2** Java ByteStream classes – classification

InputStream Classes	OutputStream Classes
ByteArrayInputStream	ByteArrayOutputStream
FileInputStream	FileOutputStream
PipedInputStream	PipedOutputStream
FilterInputStream: 	FilterOutputStream: 
ObjectInputStream	ObjectOutputStream

**Table 21.3** Members of java inputstream/outputstream classes

InputStream Methods	OutputStream Methods
<p>read() : Read a byte</p> <p>read( byte[ ] a) :Read an array b</p> <p>read( byte[ ] b , int n ,int s ) : Read an array b n bytes from s.</p>	<p>write():write a byte</p> <p>write ( byte[ ] a) : write an array b</p> <p>write ( byte[ ] b , int n ,int s ) : write an array b n bytes from s.</p>
available() : Returns no of bytes	close( ) : closes output stream file
skip(m) : Skips m bytes from input stream	flush( ) : flushes output stream file
<p>reset() : Goes to beg of file</p> <p>close() : Closes the stream file</p>	-----

### 21.3.2 Classification of *CharacterStream* Classes or *Text* Classes

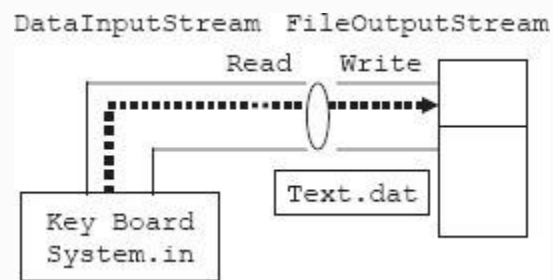
The data is represented by character, i.e., two bytes. They are best suited to handle all text files. The classification of `CharacterStream` classes are shown in Table 21.4.

**Table 21.4** Java characterstream classes – classification

Reader	Writer
<code>BufferedReader</code>	<code>BufferedWriter</code>
<code>CharArrayReader</code>	<code>CharArrayWriter</code>
<code>FilterReader</code> :	<code>Filter Writer</code> :
<code>PushbackReader</code>	<code>PrintWriter</code>
<code>InputStreamReader</code>	<code>OutputStreamWriter</code>
<code>FileReader</code>	<code>FileWriter</code>
<code>PipedReader</code>	<code>PipedWriter</code>
<code>StringReader</code>	<code>StringWriter</code>

Note that `DataInputStream` is a very useful class for reading primitive data. It

extends to `FilterInputStream` and implements `DataInput`. Accordingly it supports: `readShort()`, `readInt()`, `readDouble()`, `readLong()`, `readLine()`, `readLong()`, `readFloat()`, `readChar()`, `readBoolean()`, and a unified text format called `readUTF()`. Refer to [Figure 21.5](#).



**Figure 21.5** Input from keyboard output to a file using `FileOutputStream`

Note further that `DataOutputStream` is a class for outputting primitive data. It extends to implements `DataOutput`. Accordingly, it supports `writeShort()`, `writeInt()`, `writeDouble()`,

`writeLong()`, `writeLine()`,  
`writeFloat()`, `writeChar()`,  
`writeBoolean()`, and a unified text  
format called `writeUTF()`.

### *21.3.3 DataInputStream and DataOutputStream FileInputStream and FileOutputStream*

`DataInputStream` of `InputStream` class  
and `FileOutputStream` of  
`OutputStream` class are shown in Figure  
21.5. In this problem, the user can enter  
statements line by line. When he wants to  
stop, he can enter a dot to stop the program.  
Be advised and remember that  
`FileInputStream` and  
`FileOutputStream` handle byte bases (8  
bits) data. `FileReader` and `FileWriter`  
handle 16 bit, i.e., character data type.

**Example 21.1: CharByChar.java A  
Program to Copy a File Character by**

## Character. It Uses `FileInputStream` and `FileOutputStream`

Let us attempt a problem. We will read character by character and write on to a file called `File1`. Then we would copy the content of `File1` to `File2` character by character. Display the copied file `File2`.

```
1. package com.oops.chap21;
2. import java.io.*;
3. public class CharByChar {
4. public static void main(String[]
args) throws IOException{
5. DataInputStream input = new
DataInputStream(System.in);
6. FileOutputStream fos1 = new
FileOutputStream("File1.dat");
7. FileOutputStream fos2 = new
FileOutputStream("File2.dat");
8. System.out.println("Enter text <
enter . at end>");
9. int ch;
10. while( (ch=
(char)input.read())!='.') {
11.     fos1.write(ch); }
12.     fos1.close();
13.     // copy fos1 to fos2 character
```

by character

```
14.    FileInputStream fis = new
FileInputStream("File1.dat");
15.    System.out.println("Copyinf
from File1 & File2....");
16.    while((ch=fis.read())!=-1)
17.        fos2.write(ch);
18.    fis.close();
19.    fos2.close();
20.    //Now open the fos2 i.e.
File2.dat and display the result
21.    fis=new
FileInputStream("File2.dat");
22.    System.out.println("Displaying
the details from File2....");
23.    while((ch=fis.read())!=-1)
24.        System.out.println((char)ch);
25.    }// end of main
26.    }// end of class CharByChar
```

**Output :** Enter text < enter . at end>  
I Love India!  
I love World.  
Copyinf details from File1 & File2....  
Displaying the details from File2....  
I Love India!  
I love World

<b>Lin</b>	creates object of DataInputStream input
------------	---



<b>e No. 5:</b>	and attaches the keyboard ( <code>System.in</code> )
<b>Lin e No s. 6 &amp; 7:</b>	<code>FileOutputStream fos1/fos2 = new FileOutputStream ("File1/2.dat");</code> creates object for <code>FileOutputStream</code> and names the file as <code>File1/2.dat</code> .
<b>Lin e No s. 8– 11:</b>	reads character from key board and write to file <code>File1.dat</code> . This process continues till user enters a dot (.)
<b>Lin e No. 12:</b>	closes <code>fos1</code> .
<b>Lin e No. 14:</b>	creates object called <code>fis</code> for reading from <code>File1</code> : <code>FileInputStream fis = new FileInputStream("File1.dat");</code>
<b>Lin e No s. 16</b>	reads from object <code>fis</code> (i.e. <code>File1.dat</code> ) and copies to <code>fos2</code> (i.e. <code>File2.dat</code> ). Line No. 23 and 24 display the <code>File2</code> contents.

&  
17:

### 21.3.4 *FileReader and FileWriter*

In the example so far we have handled, we have used Byte-based IO, i.e., `FileInputStream` and `FileOutputStream` and they use 8 bits (1 byte). Our next example shows copying a file using character streams that use 16 bits (2 bytes), i.e., **`FileReader` and `FileWriter`**. Note that `FileReader` converts 8 bit character to 16 bit UTF character. The 16 bit character will be returned as `int`. So there will be no difference between the usage. Only the underlying representation of the character is different:

#### **Example**

**21.2: `FileReaderWriter.java` A Program to Copy a File Using**

## Character Streams. Obtain the Input and Output File Names Interactively from Keyboard. Use Scanner Class

```
1. package com.oops.chap21;
2. import java.io.*;
3. import java.util.Scanner;
4. public class FileReaderWriter {
5. public static void main(String[]
args) {
6. String filename1,filename2;
7. Scanner scn=new
Scanner(System.in);
8. DataInputStream input = new
DataInputStream(System.in);
9. System.out.println("Enter output
FileName :");
10. filename1=scn.next();
11. System.out.println("Enter
FileName for copy file :");
12. filename2=scn.next();
13. try{
14. FileWriter fw1 = new
FileWriter(filename1);
15. FileWriter fw2 = new
FileWriter(filename2);
16. System.out.println("Enter text <
```

```

enter . at end>");
17.  int ch;
18.  while( (ch=
(char)input.read())!='.') {
19.  fw1.write(ch);
20.  }
21.  fw1.close();
22.  FileReader fr1 = new
FileReader(filename1);
23.  while((ch=fr1.read())!=-1)
24.  fw2.write(ch);
25.  fr1.close();
26.  fw2.close();
27.  //Now open filename3 and dispaly
the result
28.  fr1=new FileReader(filename2);
29.  System.out.println("Displaying
the details from "+filename2);
30.  while((ch=fr1.read())!=-1)
31.  System.out.println((char)ch);
32.  }catch(IOException e){}
33.  }
34.  }

```

**Output :** Enter output FileName  
:citynames

Enter FileName for copy file  
:citnamescopy

Enter text < enter . to end>  
Hello India.

Displaying the details from  
citnamescopy  
Hello India

<b>Line No. 6:</b>	defines <code>filename1</code> and <code>filename2</code> as String data type to accept the file names interactively from the user.
<b>Line No. 7:</b>	defines a Scanner object <code>scn</code> to read the data from keyboard.
<b>Line No. 8:</b>	defines <code>DataInputStream</code> object and attaches keyboard ( <code>System.in</code> )
<b>Line Nos. 9–12:</b>	Accept file names from the user.
<b>Line No. 14</b>	create objects <code>fw1</code> & <code>fw2</code> for <code>FileWriter</code> class.

<b>&amp; 15:</b>	
<b>Line Nos. 16– 21:</b>	are statements for reading from keyboard and writing on to output file denoted by filename1.
<b>Line Nos. 21:</b>	FileReader fr1 = new FileReader(filename1); creates an object fr1 for filename1. Line Nos. 23–26 copies the content from filename1 to filename2.
<b>Line Nos. 28 – 31:</b>	display the content of copied file filename2.

## 21.4 IO Errors and Exceptions

Basic IO operations are likely to throw exceptions. All checked IO exceptions have to be handled by the user using try and catch

blocks. If any exception is not being handled, then the user has to throw the exception like: `public static void main(String [] args) throws IOException`. The exceptions likely are listed in Table 21.5.

**Table 21.5** IO Errors and exceptions

Input IO	Remarks
EOFException	End of file marker
FileNotFoundException	File not found
InterruptedIOException	IO interrupted
IOException	General IO Exception

## 21.5 FilterInputStream and FilterOutputStreams

Filter streams basically alter the output to suit a specified need. For example, it can be wrapped around `DataInputStream` or

BufferedInputStream. The steps involved in using FilterStream are as follows:

- Create Stream associated with data source and data destination.
- Attach suitable Filter Stream with the stream created.
- Finally use FilteredStream object to read or write data rather than original stream.

The following examples make things clear:

---

```
1 DataInputStream input = new
DataInputStream(System.in);
while( (ch=(char)input.read()) != '.' ) {
```

**Note that DataInputstream has been derived from filterInput-stream**

```
2. FileOutputStream fileout = new
FileOutputStream("DisDos");
DataOutputStream dosfile= new
DataOutputStream(fileout);
```

---

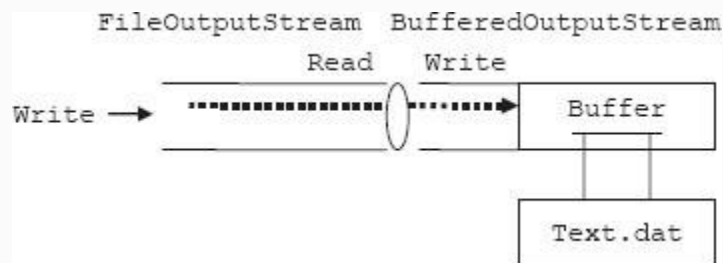
## 21.6 Using BufferedInput and BufferedOutput Streams

CPU is a fast device and memory is a slow device compared to CPU. Therefore, when we are trying to copy character on to file, the Operating System is called to coordinate the

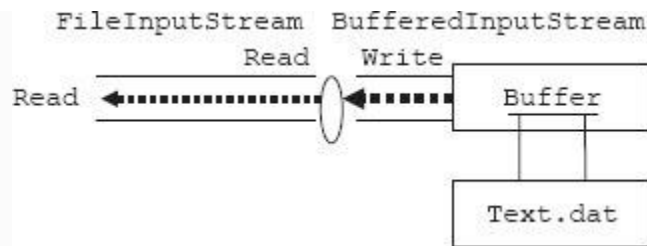


transfer from CPU to memory. This is slow and inefficient. Instead if we can provide a buffer, a temporary memory space, so that the system can write on to buffer and once buffer is filled, then OS writes to memory all at once. This is the Producer–Consumer problem we have discussed in [Chapter 20](#).

Java provides this functionality through `BufferedInputStream` and `BufferedOutputStream`, thus enhancing the efficiency of IO operations. [Example 21.4](#) deals with the method of writing data using `BufferedOutputStream` and reading data using `BufferedInputStream`. [Figure 21.6a](#) and [21.6b](#) shows the concepts involved.



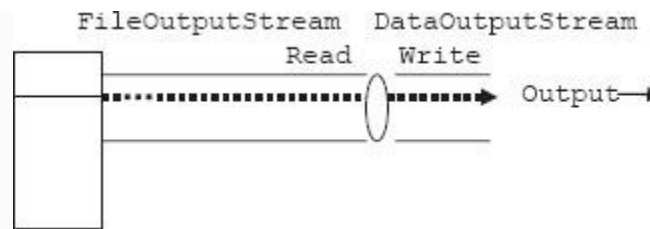
**Figure 21.6a** `BufferedOutputStream`



**Figure 21.6b** `BufferedInputStream`

## 21.7 Writing Primitive Data Types to File: `DataInputStream/DataOutputStream`

`FileInputStream`, `FileOutputStream`, `FileReader` and `FileWriter` classes are provided by Java to cater to byte and character types. But what do we do in case we want input and output of basic primitive data types like `int`, `char`, `float`, etc. `DataInputStream` and `DataOutputStream` wrap `FileInput` and `FileOutputStreams` and allow us to deal with basic data types. Figure 21.7 shows the concept involved for `DataOutputStream`:



**Figure 21.7** DataOutputStream

## Example 21.3: DisDos.Java A Program to Show Use of DataInputStream and DataOutputStream Using Scanner Class

```
1. package com.oops.chap21;
2. import java.util.*;
3. import java.io.*;
4. import javax.swing.JOptionPane;
5. public class DisDos {
6.     public static void main(String[]
args) throws IOException{
7.         int idNo;
8.         double credits,debits,netPay;
```

```

    9. System.out.println("Enter Emp
idNo,credits,debits separated by
spaces");
    10. Scanner scn = new
Scanner(System.in);
    11. idNo=scn.nextInt();
    12. credits =scn.nextDouble();
    13. debits=scn.nextDouble();
    14. netPay=credits-debits;
    15. FileOutputStream fileout = new
FileOutputStream("DisDos");
    16. DataOutputStream dosfile= new
DataOutputStream(fileout);
    17. dosfile.writeInt(idNo);
    18. dosfile.writeDouble(credits);
    19. dosfile.writeDouble(debits);
    20. dosfile.writeDouble(netPay);
    21. dosfile.close();
    22. fileout.close();
    23. //Now read data from DisDos file
    24. FileInputStream filein = new
FileInputStream("DisDos");
    25. DataInputStream disfile= new
DataInputStream(filein);
    26.
OptionPane.showMessageDialog(null," Id
Number :" +disfile.readInt() +"\nCredits
:" +
    27. disfile.readDouble()+"\t Debits
" + disfile.readDouble() +"\t Net Pay
:"+
    28. disfile.readDouble(),"Employee

```

```
Pay Bill", JOptionPane.PLAIN_MESSAGE);  
29.    }  
30.    }
```

**Output:** Enter Emp idNo, credits, debits  
separated by spaces

50595 25000.00 5000.00

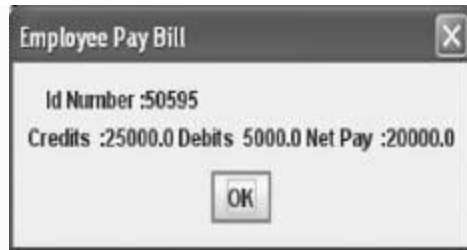
---

**Lin  
e  
No  
s.  
10  
–  
14:**

capture input data from keyboard using  
scanner class.

**Lin  
e  
No  
s.  
15–  
16:**

wrap around DataOutputStream object  
dosfile on to FileOutputStream object  
fileout created at Line No. 15. This is how  
FilterOutputStream works.



<b>Line Nos. 24–25:</b>	similarly use <code>DataInputStream</code> wrapped around <code>FileInputStream</code> .
<b>Line Nos. 26–28:</b>	use swing components to display output through message box.

## 21.8 File Class

File class helps us in creating files and directories. It has all housekeeping methods related to files such as creating, opening, closing, deleting , getting name and size of the file, renaming the file, etc.

We have been using `FileInputStream` and `FileOutputStream` by throwing `IOException`. Program comes to a halt. But

is there a better way to check if the file could be opened or not? File class and object of File can test if the file is actually allocated and other jobs such as listing of directories, etc. A file is created using a string file name and can then be tested to see different methods offered by File class. The methods offered by File class are shown in Table 21.6.

**Table 21.6** File methods

Method	Functionality
--------	---------------

<code>boolean isFile()</code>	Returns true if it is a file
<code>boolean isDirectory()</code>	Returns true if it is a directory
<code>boolean canRead()</code>	Returns true if file object is readable
<code>boolean canWrite()</code>	Returns true if file object is writable
<code>boolean exists()</code>	Returns true if file/dir exists
<code>String getParent()</code>	Returns parent directory
<code>String getPath()</code>	Returns path
<code>String getAbsolutePath()</code>	Returns absolute path from root dir
<code>long length()</code>	Returns size of file in bytes
<code>boolean delete()</code>	Deletes the file
<code>boolean createNewFile()</code>	Creates a new file if file does not exist
<code>boolean mkdir()</code>	Makes directory
<code>boolean rename(File filename)</code>	Renames the file
<code>String[] list()</code>	Returns a list of file and directories



## Example 21.4: FileDemo.Java A Program to Show Use Java's File Class

```
1. package com.oops.chap21;
2. import java.io.*;
3. public class FileDirectory {
4. public static void main(String[]
args) throws IOException{
5. System.out.println(" Getting File
Details.....");
6. File fobj = new
File("/", "Oopsjava/workspace/oopstech2/s
rc/com/chap21/DisDos.java");
7. System.out.println("File Name :" +
fobj.getName());
8. System.out.println("File Absolute
path :" + fobj.getAbsolutePath());
9. System.out.println("Parent:" +
fobj.getParent());
10. System.out.println("File exists
or not:" + fobj.exists());
11.
```

```

if(fobj.isFile())System.out.println("
Length " + fobj.length());
    12. System.out.println("is
directory?" + fobj.isDirectory());
    13. System.out.println(" Getting
Directory Details.....");
    14. File fobj2 = new
File("/", "Oopsjava/workspace/oopstech2/s
rc/com/chap21");
    15. if(fobj.exists())
    16. { // get the listing of a
directory into a array
    17. String arrList[] = fobj2.list();
    18. int len=arrList.length;
    19. System.out.println("Displying the
Directory Listing.....");
    20. for ( int i=0; i<len;i++)
    21. { System.out.println(arrList[i]);
    22. File ftemp=new
File("/Oopsjava/workspace/oopstech2/src/
com/chap21",arrList[i]);
    23. if ( ftemp.isFile())
    24. System.out.println("File :"+
ftemp.getName() + "length :" +
ftemp.length());
    25. else System.out.println("***");
    26. }//end of for
    27. }// end of if
    28. }//end of main
    29. }//end of class
Output: Getting File Details.....
File Name :DisDos.java

```

```
File Absolute path
:C:\Oopsjava\workspace\oopstech2\src\com
\chap21 \DisDos.java
```

```
Parent:\Oopsjava\workspace\oopstech2\src
\com\chap21
```

```
File exists or not:true
Length 1127
is directory?false
Getting Directory Details.....
Displying the Directory Listing.....
FOS.java
File :FOS.javalength :715
CharByChar.java
File :CharByChar.javalength :976
ByteByByte.java
File :ByteByByte.javalength :178
```

<b>Line 6:</b>	shows the creation of File object: Observe that we have to give full directory of file such as: File fobj = new File("/", "Oopsjava/workspace/oopstech2/src/com/chap21/DisDos.java");
----------------	---

<b>Line</b>	show usage of several File class method such as isFile(), isDirectory() And
-------------	--

<b>N os . 7 - 12 :</b>	<code>getAbsolutePath()</code> , and <code>getparent()</code> , etc.
<b>Li n e N o. 1 0:</b>	checks if specified file or directory exists by: <code>fobj.exists()</code> method.
<b>Li n e N o. 17 :</b>	<code>String arrList[] = fobj2.list();</code> gets the files and directories and allocates them to <code>String arrList[]</code>
<b>Li n e N o. 21 :</b>	attaches <code>arrList[i]</code> with a File object and gets length and name at Line No. 24.

---

## 21.9 Random Access Files

In this mode of access, a record is accessed using index maintained for this purpose. It is like going to a chapter and within the chapter to a page of interest, using the Index provided at the end of the book. The file can be accessed from any location. It implements both `DataInput` and `DataOutput` interfaces and extends to Java's base class `Object`. The syntax for using random files in Java is

---

```
RandomAccessFile( File obj , String  
Access Specifier);  
RandomAccessFile( String filename ,  
String Access Specifier);
```

---

Access specifier is a mode that determines the mode of operation of Random Access File. "`r`" indicates for read only and "`rw`" indicates both for read and write. Further, there is a method called `seek()` that can be used to set the current position in the file. `Seek()` method justifies the name Random

**File.** The syntax for `seek()` is: `void seek(long pos)` throws `IOException`.

There is also a provision to set the length of the Random file: `void setlength(long n)` throws `IOException`, sets the length of random files. If there are additional entries, they are ignored.

### **Example 21.5: RandomAccess.Java** **A Program to Show Use Java's File Class**

```
1. package com.oops.chap21;
2. import java.io.*;
3. public class RandomAccess {
4. public static void main(String[]
args) throws IOException {
5. //create the object for RandomFile
6. RandomAccessFile randfile = new
RandomAccessFile("raf.dat ", "rw");
7. try {
8. int idNo=50595; float sal
=20000.00F; boolean malefemale= true;
```

```
9.  char cadre = 'S';
10.  randfile.writeInt(idNo);
11.  randfile.writeDouble(sal);
12.  randfile.writeChar(cadre);
13.  // go to beginning of file
14.  randfile.seek(0);
15.  System.out.println("idno :"+
randfile.readInt());
16.  System.out.println("Salary :"+
randfile.readDouble());
17.  System.out.println("Cadre :"+
randfile.readChar());
18.
randfile.seek(randfile.length()); //go to
end of file
19.
randfile.writeBoolean(malefemale);
20.  randfile.seek(3);
21.  System.out.println("Malefemale
:"+ randfile.readBoolean());
22.  randfile.close();
23.  }catch( IOException e){}
24.  } // end of main
25.  } // end of class
Output : id no :50595
Salary :20000.0
Cadre :S
Malefemale :true
```

---

<b>Line No. 6:</b>	<code>RandomAccessFile randfile = new RandomAccessFile("raf.dat", "rw") ; opens random file in read write mode.</code>
<b>Line Nos. 10–12:</b>	<code>write int, char an double data on to Random file by randfile.write method</code>
<b>Line Nos. 15–17:</b>	<code>display the data written on to random file by read command.</code>
<b>Line No. 18:</b>	<code>randfile.seek(randfile.length()) ; // go to end of file</code>

## 21.10 Serialization of Objects and Object Streams

Serialization means storing objects on to file. De-serialization means the reverse of serialization and it means getting objects from file to class object in memory. The file to be serialized has to implement an interface called `Serializable`. Storing an



object in a file and reading back object involves object Streams such as `ObjectOutputStream` and `ObjectInputStream`. The steps involved are shown below.

### *21.10.1 Serialization*

<b>Step 1:</b>	define a class & create an object class <code>Student implements Serializable{private : idNo; String name;} Student std=new Student()</code>
<b>Step 2:</b>	Use <code>FileOutputStream</code> : <code>FileOutputStream fosobj= new FileOutputStream("std.dat")</code>
<b>Step 3:</b>	Use <code>ObjectOutputStream</code> : <code>ObjectOutputStream objoutstream = new ObjectOutputStream(fosobj);</code>
<b>Step 4:</b>	Write on to file using <code>writeObject()</code> method of <code>FileOutputStream</code>

Note that Serializable interface has no methods at all. It is used by Java to mark the file as Serializable, i.e., file to be written on to file. Static variables cannot be serialized.

### *21.10.2 De-serialization*

<b>Step 1:</b>	define a class & create an object class Student implements <code>Serializable{private : idNo; String name;} Student std=new Student()</code>
<b>Step 2:</b>	Use FileInputStream: <code>FileInputStream fisobj= new FileInputStream("std.dat")</code>
<b>Step 3:</b>	Use ObjectInputStream: <code>ObjectInputStream objinstream = new ObjectInputStream(fisobj)</code>
<b>Step 4:</b>	Read file using <code>readObject()</code> method of <code>FileInputStream Student std = ( Student) objinstream.readObject()</code>

## Example

### 21.6: SerializableDemo.Java A Program to Show Serialization and Deserialization of Objects

```
1. package com.oops.chap21;
2. import java.io.*;
3. class Student implements
Serializable {
4. // member data or attributes
5. private int idNo = 50595;
6. private double totalMarks=990.0;
7. private String grade="A";
8. public double getMarks(){ return
totalMarks;}
9. public int getIdNo(){ return
idNo;}
10. public String getGrade(){ return
grade;}
11. public void DisplayData(Student
std) {
12. System.out.println("Details from
Student are :");
13. System.out.println("idNo:" +
std.getIdNo());
```

```

    14. System.out.println("Total Marks:"
+ std.getMarks());
    15. System.out.println("Grade:" +
std.getGrade());
    16. }//end DisplayData
    17. }// end of class
    18. // Driver class
    19. class SerializableDemo {
    20. public static void main(String[]
args)throws IOException{
    21. // create an object of Student
    22. Student std = new Student();
    23. System.out.println("Displaying
Data from Student class....");
    24. std.DisplayData(std);
    25. FileOutputStream fosobj= new
FileOutputStream("Stdobj.dat");
    26. ObjectOutputStream objoutstream =
new ObjectOutputStream (fosobj);
    27. objoutstream.writeObject(std);
    28. objoutstream.close();
    29. System.out.println("Reading from
Serialized Object Student...");
    30. FileInputStream fisobj= new
FileInputStream("Stdobj.dat");
    31. ObjectInputStream objinstream =
new ObjectInputStream (fisobj);
    32. try{
    33. std =
(Student)objinstream.readObject();
    34. }catch(ClassNotFoundException e)
{}

```

```

35. std.DisplayData(std);
36. objinstream.close();
37. }
38. }//end of class Student2bDemo

```

**Output** : Displaying Data from Student class....

```

Details from Student are : idNo:50595
Total Marks:990.0 Grade:A
Reading from Serialized Object

```

Student...

```

Details from Student are : idNo:50595
Total Marks:990.0 Grade:A

```

<b>Line No . 6:</b>	shows the Student file implementing Serializable interface. This implantation is mandatory.
<b>Line No . 25:</b>	FileOutputStream fosobj= new FileOutputStream ("Stdobj.dat"); Creates FileOutputStream object.
<b>Line No</b>	ObjectOutputStream objoutstream = new ObjectOutputStream(fosobj); <b>creates object for ObjectOutputStream.</b> <b>This is FilterOutputStream at work.</b>

<b>. 26:</b>	
<b>Lin e No . 27:</b>	<code>objoutstream.writeObject(std);</code> writes to file
<b>Lin e No s. 30 – 35:</b>	show the reverse, i.e., de-serialization.

## 21.11 Summary

1. IO streaming means flow of data from input device to memory and vice versa.
2. Buffering is used whenever movement of data is between the devices with large differences in their speeds of operation.
3. For reading from keyboard to computer system, `InputStreamReader` and `BufferedReader` are used. `DataInputStreamReader` accepts the data from keyboard and `System.in` represents keyboard.
4. `System.in`: Standard input is from keyboard. This represents `InputStream` Object

5. `System.out`: Console is represented by `System.out`. This represents `PrintStream` object.
6. `System.err` is used to log errors that occur while handling streams.
7. `File` is a collection of records. A record, in turn, contains fields.
8. `Java.io` package contains stream classes for handling File IO. Stream Files can be divided into two types: `ByteStream` classes and `CharacterStream` classes. These Stream classes read from source and write to destination. The source and destinations are: Memory, pipe and File.
9. Both `InputStream` and `OutputStream` classes are derived from Java's `Object` class.
10. `ByteStream` classes are `FileInputStream`, `FileOutputStream`, `BufferedInputStream`, and `BufferedOutputStream`.
11. `CharacterStream` Classes are `FileReader`, `FileWriter`, `BufferedReader`, and `BufferedWriter`.
12. `ByteStream` Classes. The data is represented as byte (8 bits). All text files and audio and video files that need to be transmitted using Internet are stored using `ByteStream` Classes.
13. `CharacterStream` Classes. The data is represented by character, i.e., two bytes. They are best suited to handle all text files.
14. `DataInputStream` is a very useful class for reading primitive data. It extends to `FilterInputStream` and implements `DataInput`.
15. `DataOutputStream` is a class for outputting primitive data. It extends to implements `DataOutput`.

16. `FileInputStream` and `FileOutputStream` handle byte bases (8 bits) data.
17. `FileInputStream fis = new  
FileInputStream("File1.dat");`
18. `FileOutputStream fos = new  
FileOutputStream("File1.dat");`
19. `FileReader` and `FileWriter` handle 16 bit, i.e., character data type. `FileReader` converts 8-bit character to 16-bit UTF character. The 16 bit character will be returned as `int`.
20. All checked IO exceptions have to be handled by the user using try and catch blocks. If any exception is not being handled, then the user has to throw the exception like `public static void main(String [] args) throws  
IOException.`
21. Errors and Exceptions during `InputStream` operations are `EOFException`, `FileNotFoundException`, `InterruptedIOException`, and `IOException`.
22. `BufferedInputStream` and `BufferedOutputStream` enhance the efficiency of IO operations.
23. `DataInputStream` and `DataOutputStreams` wrap `FileInput` and `FileOutputStreams` and allow us to deal with basic data types.
24. `File` class helps us in creating files and directories. It has all housekeeping methods related to files such as creating, opening, closing, deleting, getting name and size of the file, renaming the file, etc.
25. Random Access Files. The access to file can be from any location. It implements both `DataInput` and `DataOutput` interfaces and extends to Java's base class `Object`.



26. Access specifier is a mode that determines the mode of operation of Random Access File. "r" indicates for read only and "rw" indicates both for read and write.
27. A method called `seek()` that can be used to set current position in the file. `Seek()` method justifies the name Random File. The syntax for `seek()` is :`void seek(long pos)` throws `IOException`.
28. Serialization means storing objects on to file. De-serialization means reverse of serialization and it means getting objects from file to class object in memory. The file to be serialized has to implement an interface called `Serializable`.
29. Storing an object in a file and reading back object involves object Streams such as `ObjectOutputStream` and `ObjectInputStream`.

## Exercise Questions

### Objective Questions

1. Which of the following statements are true?

1. The smallest unit a file can handle is called bit.
2. Byte Stream handles IO streaming with 16 bits.
3. Character Stream handles data with 16 bits.
4. Text files are best handled by byte stream.

1. i and iv
2. iii and iv
3. i and ii
4. i and iii

2. Which of the following statements are true in respect of IO Stream files?

1. `InputStream` and `OutputStream` are descendants of `Object` class.

2. IO Stream can handle threads.
3. `BufferedReader` is used to handle byte data.
4. Buffered Streams enhance the efficiency of file handling.

1. i and iv
2. iii and iv
3. i, ii and iv
4. i and iii

3. Which of the following statements are true in respect of `DataInput/DataOutput` class

1. Filter streams basically alter the output to suit a specified need
2. `dosfile.writeString(stg)` is a valid statement
3. `FilterOutputStream` can be wrapped around `DataOutputStream`
4. `DataInputStream` is derived from `FileInputStream`

1. i, ii and iv
2. i, iii and iv
3. i and iii
4. ii, iii and iv

4. Which of the following statements are true in respect of `File` class?

1. File object can be used to check if file exists
2. `String getPath()` gets the path from root
3. `String[] list()` lists list of files and directories
4. `boolean createNewFile()` overwrites if file already exists

1. i, ii and iv
2. i and iii
3. i and iii
4. ii, iii and iv

5. Which of the following statements are true in respect of Random Access File?

1. `readInt()` is a valid method to read input from specified stream
2. `seek(m)` sets the current position starting at `m+1` byte
3. "a" opens Random file in append mode
4. Random access file can have variable length records

1. i, ii and iv
2. i and iii

- 3. i and iii
- 4. ii, iii and iv

6. A random access file if opened in "rw" mode can be used for append by using statement `seek(fin.length())` where fin is the object of Random Access File. TRUE/FALSE

#### Short-answer Questions

- 7. What are files, records and fields?
- 8. What is IO Stream?
- 9. How are Streaming classes classified?
- 10. What types of files are best suited for `ByteStream` classes?
- 11. What types of files are best suited for `CharacterStream` classes?
- 12. What are `FileReader` and `FileWriter`?
- 13. What are the classes used for input and output of primitive data types?
- 14. What are the modes of operation of Random Access File?
- 15. What are serialization and de-serialization?

#### Long-answer Questions

- 16. Explain IO Streaming classes with suitable examples.
- 17. Distinguish `InputStream` and `Reader` class and `OutputStream` and `Writer` classes.
- 18. Explain the utility of `File` class.
- 19. Explain the classification of Byte Stream classes.
- 20. Explain the classification of Character Stream classes.
- 21. Explain errors and exceptions while handling files with suitable examples.
- 22. Explain how end of file marker is checked in Java.
- 23. How can we read primitive data from keyboard using scanner class and `DataInputStream`? Explain with suitable code segments.

24. Explain the utility of `BufferedReader` and `BufferedOutputStream`.
25. Explain Serialization and De-serialization Process.

#### **Assignment Questions**

26. Write an Inventory Control Program with Random Access file with features to `addItem()`, `issueItem()` and `updateItem()`.
  27. Write a java program to implement Telephone Directory wherein given the name, telephone number is output. Use Object serialization concept for writing on to file.
  28. Write a program to simulate Attendance and Performance Query System which answers students query on attendance and performance. Assume random access organization.
  29. Develop java program to zip a text file using `InflaterInputStream` and `deflatorOutputStream`. Hint: use wrapping around as shown below for Deflator
- 

```
FileOutputStream fileout = new
FileOutputStream("data");
DataOutputStream dosfile= new
DataOutputStream(fileout);
```

---

30. Write a java program to convert an input text file to upper case file. Use
31. `Character.toUpperCase()` method.

#### **Solutions to Objective Questions**

1. b

2. c

3. c

4. b

5. c

6. True

# 22

## Networking in Java

### LEARNING OBJECTIVES

*At the end of the chapter, you will be able to understand and use*

- TCP/IP protocols.
- Inet addressing mechanism and URL connections.
- TCP/IP sockets.
- Client server applications.

### 22.1 Introduction

Java has been specially created for networking. What do we mean by this? Java supports a package called `java.net` which

defines several methods for connecting and retrieving files by using common web-based protocols. Java also allows us to create unix-like sockets and allows us to implement TCP/IP sockets.

In the previous chapter on Java IO programming, you have been introduced to Input and Output Stream. By attaching those stream files with networking protocols, you can write to files on the Internet and retrieve data from files over the Internet as if you are writing files on to a disk.

This chapter covers the basics of networking such as TCP/IP protocols and its implementation through TCP/IP and datagrams. URL connections and Inet addressing mechanisms are also presented. Client server application employing TCP/IPs and sockets are discussed.

## 22.2 Basics of Networking

Networking is the ability to connect to the Internet from our stand-alone applications or Applets. The classes required for this

interconnection are available in java.net package. This package provides methods for creating sockets and file retrieval from any other networked cooperating system using web protocols. By attaching streaming classes with these network socket classes, writing and reading data to and from the Internet becomes easy.

There are two terms closely connected with java programming: Internet and World Wide Web (www). They are NOT synonymous, and they are NOT the same.

**Network** is a collection of computers that is either homogenous or heterogeneous.

The **Internet** is a collection or grouping together of networks. We can say that the Internet is a network of networks, comprising of millions of computers. These computers communicate with each other using Internet protocols.

**World Wide Web (WWW)** is a way of accessing information on the Internet. It uses HTTP to exchange data amongst communicating computers. The services provided by www uses Internet browsers such as Internet Explorer (IE), FireFox,



Google Chrome, etc. The information is exchanged using HTTP and web pages. Web page is a place where required information is stored. Web pages are linked through hyperlinks. www is a method of Internet to exchange data over Internet.

Computers running on the Internet communicate with each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

### *22.2.1 TCP/IP*

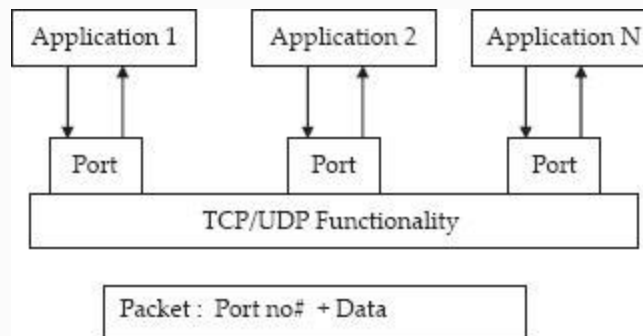
A protocol is a set of rules to be followed by cooperating systems on the Internet for transferring data from one system to another system. TCP/IP, i.e., Transmission Control Protocol and Internet Protocol, are standard protocols to be used for transmission of data.

TCP/IP is a connection-oriented and reliable service.

- Connection oriented means the sender confirms if the receiver has indeed received the data or not. If not received, then the retransmission of data takes place.
- TCP/IP is a point-to-point reliable communication, i.e., it is an acknowledged service that ensures correct

reception of data at receiver. TCP/IP has five layers, as shown in [Figure 22.3](#).

- Java programs written come under Application Layer. Hyper Text Transfer Protocol (HTTP) and File Transfer Protocol (FTP) and Telnet are some of the widely used protocols at the application layer.
- Application Layer receives the data from IO streams of sender and format data as streams of continuous bytes and sends them to Transport layer, as shown in [Figure 22.1](#).



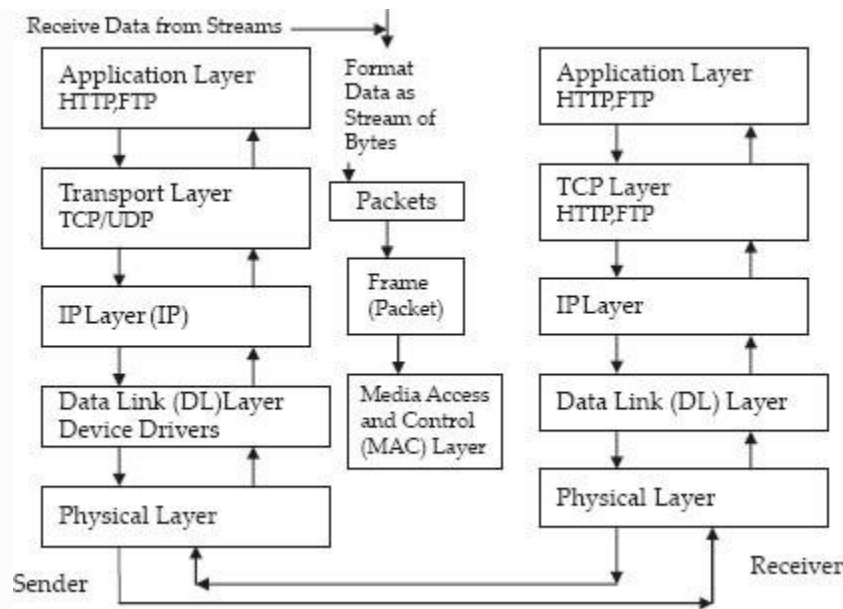
**Figure 22.1** TCP/UDP functionality

- Transport Layer receives these streams of data through assigned ports and makes a packet ([Figure 22.1](#)). A packet will encompass Port Number and data.
- Port numbers vary between 0 and 65535 and are represented by a 16-bit number. Note that these numbers are logical and have nothing to do with physical ports on computer systems. The port numbers ranging from 0 to 1,022 are restricted; they are reserved for use by well-known services such as HTTP and FTP and other system services. Examples of well-known ports are:
  - Domain Name System (DNS): 53
  - File Transfer Protocol (FTP): 21
  - Hypertext Transfer Protocol (HTTP): 80

- Network News Transfer Protocol (NNTP):119
  - Post Office Protocol (POP3): 110
  - Simple Mail Transfer Protocol (SMTP): 25
  - Telnet: 22
- IP Layer receives the packet and encapsulates the packet into envelopes called frames (**Figure 22.2**). A frame will have source address, destination address, data, and some bits reserved for error detection and correction mechanism. For specifying the address, IP address is specified. An IP address is a 32-bit address which identifies computers connected to networks uniquely. For example, 192.167.169.24. This protocol is called IPV 4. A new protocol called IP V 6 has been introduced which follows 48-bit addressing mechanisms and the Internet allows both IPs.



**Figure 22.2** IP layer encapsulation of packet in a frame



**Figure 22.3** TCP/IP model layers

- Address of destination can be specified by numeric address like 192.127.68.24 or by domain name like www.google.com. The transformation is carried out by Domain name server which transforms address from numeric to web site name and vice versa.
- Data Link Layer receives the Frame from IP layer and dispatch the frame to destination computer. This layer has device drivers and contains Media Access Control Protocol in Network Card. Network card maps the IP address into 48-bit unique Network card id number and transmits based on MAC protocol. MAC protocol simply tells which is going on the Net next, i.e., it tells the rules for sharing single transmission medium by multiple frames.
- When the destination computer received the frames, the formats are decoded based on the protocol used and frames are handed over to the IP layer. The IP layer produces packets and hands it over to Transport Layer.

The transport layer hands over data at specified ports, as shown in [Figure 22.3](#).

### *22.2.2 User Data Gram Protocol (UDP)*

UDP is a connection-less service. It means communication is not guaranteed between two applications. Delivery is not important and is not guaranteed. It is mostly used in transmission of video and audio, etc., where elaborate error detection and correction are not required and little bit of data missing cannot make perceptible differences.

## 22.3 Internet Address

We can obtain Internet address of any web site by using `InetAddress` class of `java.net` package. `InetAddress` class supports the following methods:

---

```
static InetAddress getByName (String
hostName) throwsUnknownHostException
static InetAddress[] getAllByName
(String hostName)
throwsUnknownHostException returns all
addresses that a particular web site
name is representing.
```

```
static InetAddress getLocalHost()  
throwsUnknownHostException
```

---

## Example

### 22.1: InetAddressDemo.java Write a Program to Get Inetaddresses and Local Host Name for a Specified Web Site

---

```
// This program requires Internet  
connection  
1. package com.oops.chap22;  
2. import java.io.*;  
3. import java.net.*;  
4. import java.util.*; // for scanner  
class  
5. public class InetAddressDemo {  
6. public static void main(String[]  
args) throws IOException {  
7. String websitename;  
8. Scanner scn=new  
Scanner(System.in);  
9. System.out.print("Enter WebSite  
Name :");
```

```

10. websitename=scn.next();
11. try{
12. InetAddress
ipadd=InetAddress.getByName(websitename)
;
13. System.out.println(" The IP
address obtained from Internet is : "+
ipadd);
14. }catch(UnknownHostException e)
15. { System.out.println("Web Site
NOT found");}
16. }//end of main
17. }//end of class

```

OUTPUT : Enter WebSite Name

:www.google.com

The IP address obtained from Internet  
is : www.google.com/209.85. 221.104

## 2 nd Run

Enter WebSite Name

:www.vasappanavara.org

The IP address obtained from Internet  
is : www.vasappanavara.org/  
74.86.158.226

<b>Line</b>	obtains the name of the web site by calling static method <code>getByName()</code> by creating object <code>ipadd</code> to <code>InetAddress</code> class. Line No 13 displays the result.
-------------	---

<b>o. 12 :</b>	
<b>Li n e N o. 14 :</b>	catches (UnknownHostException e)

## 22.4 URL and URL Connection

Uniform Resource Locator (URL) has been created to get the information from the net, no matter which protocol has been employed to create it. We are aware that www is a collection of several protocols like HTTP/FTP/finger/whois. URL is a modern way to acquire information from the Internet. URLa class provides methodologies to access information using URL.

### URL Constructors



- `URL (String protocol, String host, int port, String file)` throws `MalformedURLException`.
- `URL (String protocol, String host, String file)` throws `MalformedURLException`.
- `URL (String urlString)` throws `MalformedURLException`.

## URL Methods

- `String getFile()`
- `String getHost()`
- `int getPort()`
- `String getProtocol()`

## URL: How to Set It to Work?

- Create URL object that represents resource on www address
- `URL myurl = new URL("http://www.vasappanavara.org");`
- Create an URL connection that can connect to concerned web site
- `InputStream inputStream = myurl.openStream();`
- Using `StreamReader` create a buffer, say of data type `byte`, for reading stream data from URL connection.  
`byte buffArray[ ] = new byte[1024];`
- Use `read()` method of that `URLConnection` object and create `InputStream` for reading stream of data from the URL.

---

```
while((i =  
inputstream.read(buffArray)) != -1) {  
    System.out.write(buffArray, 0, i);}}
```

---

These concepts are shown in our next example.

## Example

### 22.2: InetAddressDemo.java Write a Program to Get InetAddresses and Local Host Name for a Specified Web Site

---

```
//This program need Internet  
Connection  
import java.io.*;  
import java.net.*;  
import java.util.*; // for scanner  
class  
    public class URLEDemo {  
        public static void main(String[]  
args) throws IOException {  
            String websitename;  
            Scanner scn=new Scanner(System.in);
```

```

        System.out.print("Enter WebSite Name
:");
        websitename=scn.next();
        try{
            URL myurl = new URL(websitename);
            // Obtain input stream
            InputStream inputstream =
myurl.openStream();
            // Read and display data from URL
            byte buffArray[] = new byte[1024];
            int i;
            while((i =
inputstream.read(buffArray)) != -1) {
                System.out.write(buffArray, 0, i);
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

```

**Output :** Enter WebSite Name  
: http://www.vasappanavara.org  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD  
HTML 4.01 Transitional//EN"  
  
"http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
<meta http-equiv="Content-Type"  
content="text/html; charset=iso-8859-1">  
<title>Prof. Ramesh

```
Vasappanavara</title>
  <style type="text/css">
  <!--
  .....
</body>
</html>
```

---

### *22.4.1 URL Connection*

With URL class we could import the content of web site, but could not examine the content in detail before importing. URL connection is a class that allows us to examine the properties prior to transporting. Methods supported by URL connection are shown in Table 22.1.

**Table 22.1** URLConnection class methods

URLConnection Class Methods		Functionality
int getContentLength()	Returns size in bytes else -1	
long getDate()	Returns response time and date in milliseconds after Jan1 1970, of last modification	
long getExpiration()	Returns expiration time and date in milliseconds after Jan1 1970, of last modification	
long getLastModified()	Returns time and date in milliseconds after Jan1 1970, of last modification	
InputStream getInputStream() throws IOException	Returns InputStream of URL	

## URLConnection: How to Set It to Work?

- Create URL object that represents resource on www address  

```
URL myurl = new
URL("http://www.vasappanavara.org");
```
- Create an URL connection that can connect to concerned web site.

- Use `getInputStream()` method of that `URLConnection` object and create `InputStream` for reading stream of data from the URL.
- Using input stream reader, create a `BufferedReader` object for reading stream data, i.e., characters from URL connection.

**These concepts are shown in our next example.**

## Example

### 22.3: `URLConnectionDemo.java`

### Write a Program to Show Usage of `URLConnection` Class Methods

---

```
//This program need Internet
Connection
1. package com.oops.chap22;
2. import java.io.*;
3. import java.net.*;
4. import java.util.*; // for scanner
class
5. public class URLConnectionDemo {
6. public static void main(String[]
args) throws IOException {
7. String websitename; int ch;
```

```

8. Scanner scn=new
Scanner(System.in);
9. System.out.print("Enter WebSite
Name :");
10. websitename=scn.next();
11. try{
12. URL myurl = new URL(websitename);
13. URLConnection vrCon =
myurl.openConnection();
14. // get date
15. long dt=vrCon.getDate();
16. System.out.println("Date :"+ new
Date(dt));
17. // get content length
18. int len=vrCon.getContentLength();
19. System.out.println(" Size :"+
len);
20. //get last modified date
21. dt=vrCon.getLastModified();
22. System.out.println("Date 1st
modified:"+ new Date(dt));
23. //get the content
24. if (
len==0)System.out.println("NoContent");
25. else
26. { InputStream inputstream =
vrCon.getInputStream();
27. while( ( ch =
inputstream.read())!=-1)
28. System.out.print((char)ch);
29. inputstream.close();
30. }

```

```

31. }catch (Exception e)
{e.printStackTrace();}
32. }
33. }

Output: Enter WebSite Name
:http://www.vasappanavara.org
Date :Sun Dec 19 19:56:32 IST 2010
Size :16880
Date 1st modified:Fri Oct 01 07:04:33
IST 2010
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.01 Transitional//EN
.....
</body>
</html>

```

<b>Line Nos</b> • <b>12 &amp; 13</b> :	URL myurl = new URL(websitename); URLConnection vrCon = myurl.openConnection(); create an object of URL called myurl and uses openConnection() method of URLConnection class
<b>Line</b>	get information about URL such as date, content length , last modified date etc



<b>N os . 15 - 21 :</b>	
<b>Li ne N os . 26 &amp; 27 :</b>	<pre>InputStream inputstream = vrCon.getInputStream(); while( ( ch = inputstream.read()) != -1) read the input stream and prints content of web site character by character</pre>

## 22.5 TCP/IP Sockets

You would have noticed power sockets. Insert a plug and power flows through the socket and runs your devices. Likewise, a socket in communications is connecting point between a server and a client so that a reliable bidirectional communication can go on between server and client.

Sockets have unique identification numbers also called port numbers and take 16 bits (2 bytes). In Section 22.2.1, we have already discussed port numbers that are reserved for standard applications such as `http/ftp`, etc. Note that whenever a new socket is created, a new port number must be allocated to the port.

There are two kinds of sockets: Server Side socket called *ServerSocket*, created using `ServerSocket` class, and client side socket called *socket* Created using a `socket` class. A `ServerSocket` can reside in any system that has service to offer. For example, a `ServerSocket` can be created on client if it has any services to offer.

### *22.5.1 ServerSocket Class*

`ServerSocket`, once created, listens either to local or remote clients and connects them on particular published ports. This is accomplished by `ServerSocket` first registering itself with the system. The constructors accept the port number and

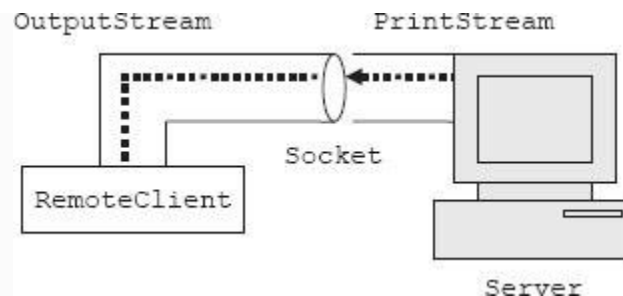
queue length, i.e., the maximum number of clients a server can accept. The constructors for `ServerSocket` are listed in Table 22.2. `ServerSocket` has a method called `accept()` that blocks the server and waits for client request for connection. A socket for the client is created by the server

**Table 22.2** `ServerSocket` constructors

ServerSocket Class Constructors		Functionality
<code>ServerSocket(int port)</code>	ServerSocket on specified port with default queue length of 50 is created	
<code>ServerSocket(int port, int len)</code>	ServerSocket on specified port with queue length of len is created	
<code>ServerSocket(int port, int len, InetAddress localaddress)</code>	ServerSocket on specified port with queue length of len is created  Belonging to local address system	

## 22.5.2 Server and Socket for Communications: How to Set them to Work?

- At server, create a server socket using `ServerSocket` class and allocate a port number: `ServerSocket servskt = new ServerSocket (port number);`
- Make the server wait for client connection request by creating a client socket by using `Socket` class and by using `accept()` method `Socket clientskt = servskt.accept();`
- **Link OutputStream to socket object clientskt just created by using `getOutputStream()` which returns a output stream object. This object will be used by clientskt to sent the data to client.**  
`OutputStream comobj = clientskt.  
getOutputStream();`
- **Use `PrintStream` to transport data to socket:**  
`clientskt  
PrintStream prntobj =new  
PrintStream(comobj)`
- **Use `println` or `print` method to send data**  
`prntobj.println(stg); //stg is a message`
- **At the end of transmission close all sockets:**  
`servskt.close(); clientskt. close(); and  
prntobj.close().` **Figure 22.4** shows interconnection of server & client & stream readers & buffers.



**Figure 22.4** Input from remote client

### 22.5.3 Client and Socket for Communications: How to Set them to Work?

- At Client, create a client socket using Socket class and specify the IPAddress of the server and allocate a port number :  

```
Socket skt = new Socket ("IPAddress",  
port number);
```

 IPAddress is the IP address of the server. To know IPAddress of a computer, goto ctrl panel → NetworkConnections --> Local area connections ---> TCP/IP ---> you can see the IPAddress of a machine.
- Link InputStream to socket object skt just created by using  

```
getInputStream()
```

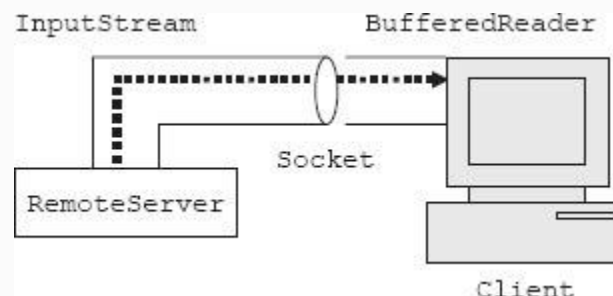
 which returns an input stream object. This object will be used by skt object to receive data from the server. InputStream comobj = skt. getInputStream();
- Use BufferedReader to get the data from socket:  

```
skt
```

- ```

BufferedReader bfrtobj = new
BufferedReader (new
InputStreamReader (comobj));

```
- Use `read()` or `readLine()` method to read the data  
`Stg= bfrobj.readLine(); // stg is a message`
  - At the end of transmission close all sockets :  
`skt.clos(); bfrobj.close();` . **Figure 22.5** shows the interconnection of server & client & stream readers & buffers.



**Figure 22.5** Input from remoteserver

## 22.6 Client Server Program

In this section, you will learn client server programming. As the heading suggests, there is a server which is running forever listening for clients on a published port. Obviously the server has a service to offer which is of interest to the client. There is a limit to the number of clients a server can accept; this limit is called max queue length

of the server. The server uses `ServerSocket` class.

Then there is a client who knows the server's address and port number on which a particular application is running. Client sends a service request which will be `accept()` method by Server. `Accept()` method also returns a `ServerSocket` object.

After the connection is established, both server and client will establish stream connections and carry out communications. As a first example, we will establish connection and pass two string messages to client.

**Example 22.4: `SrverDemo1.java` A Java Program for Server Side Docket for Passing String Messages to Client**

---

```
1. package com.oops.chap22;
2. import java.io.*;
3. import java.net.*;
4. public class SrverDemol {
5. public static void main(String[]
args)throws IOException {
6. //server socket
7. ServerSocket servskt = new
ServerSocket (8345);
8. //block the server i.e use
accept()& create client socket at server
9. Socket clientskt =
servskt.accept();
10. // Link OutputStream to socket
object clientskt with getOutputStream()
11. OutputStream comobj = clientskt.
getOutputStream();
12. //Use PrintStream to transport
datas to socket : clientskt
13. PrintStream prntobj =new
PrintStream( comobj);
14. String stg1 = "Welcome client";
15. String stg2 = " Networking
program run a great fun!";
16. //send data
17. prntobj.println(stg1);
18. prntobj.println(stg2);
19. //close all
20. servskt.close();
clientskt.close();
```



```
21. }  
22. }
```

|                              |                                                                                                                                                                                                                     |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Line No. 7:</b>           | creates a <code>ServerSocket</code> object called <code>sevskt</code> . Line No. 9 creates an object of <code>Socket</code> class for client and makes <code>ServerSocket</code> call <code>accept()</code> method. |
| <b>Line No. 11:</b>          | attaches output stream to client socket<br><code>OutputStream comobj = clientskt.<br/>getOutputStream();</code>                                                                                                     |
| <b>Line No. 12:</b>          | creates a print stream for sending data to client<br><code>PrintStream prntobj =new<br/>PrintStream( comobj);</code> Line No. 17 & 18 send the <code>String</code> message to client                                |
| <b>Line No. 17 &amp; 18:</b> | send messages to client<br><code>prntobj.println(stg1);</code>                                                                                                                                                      |

Run the program from command prompt. Refer to Example 15.5 for necessary instructions. For simplicity and ease of execution, you can remove the package class from all the client server programs and compile and execute all in one directory, say

```
c:\oopsjava\workspace\oopstech\src\co\oops\chap22
```

---

Exercise problems at the end of the chapter show the way.

### **Example 22.5: ClientDemo1.Java A Client Side Socket Program to Establish Connection with Server and Receive Messages from Server**

---

```
1. package com.oops.chap22;  
2. import java.io.*;  
3. import java.net.*;
```

```

4. public class ClientDemo1 {
5. public static void main(String[]
args)throws IOException {
6. //create a client socket using
Socket class
7. Socket skt = new Socket (
"localhost" ,8345);
8. //Link IntputStream to socket
object skt
9. InputStream comobj = skt.
getInputStream();
10. //Use BufferedReader to get the
data from socket
11. BufferedReader bfrtobj =new
BufferedReader ( new
InputStreamReader(comobj));
12. //Receive messages
13. String stg;
14. while(( stg=bfrtobj.readLine())
!= null)
15. System.out.println("\n Message
from Server" + stg);
16. //close all
17. skt.close(); bfrtobj.close();
18. }
19. }

```

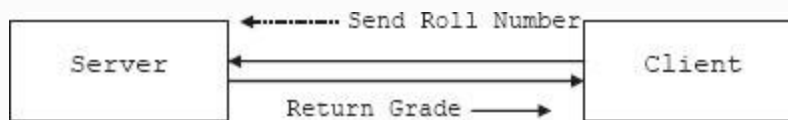
**Output:** Message from Server :Welcome  
client

Message from Server Networking  
program run a great fun!

---

### 22.6.1 Client Server Two-way Communication Program

In this section, we will handle two-way communication between a server and a client. Server stores two arrays. In one array RollNumbers are stored and in the second array academic grades are stored. Client sends roll number and server checks for the grade and sends them to client. Client then displays the result. The concept is shown in Figure 22.6.



**Figure 22.6** Client server two-way communications

**Example 22.6a: Write Java Client Server Program Wherein Client Submits Roll Number and Gets Grade from Server—Server Side Programming**

```
package com.oops.chap22;
import java.io.*;
import java.util.*;
import java.net.*;
public class ServerDemo2 {
    public static void main(String[]
args)throws IOException {
    int id, index=0;String gr;
    int []rolls = new int[]
{50595,50596,50597,50598,50599};
    String[] grade= new String[]
{"A","C","B","C","A"};
    try{
        //server socket
        ServerSocket servskt = new ServerSocket
(5000);
        //block the server i.e use accept() &
create client socket at server
        Socket clientskt = servskt.accept();
        // Create a BufferedReaderStream to get
the data from the client
        BufferedReader bufrclient=new
BufferedReader
        ( new
InputStreamReader(clientskt.getInputStre
am())));
        //create buffer writer stream for
sending data to client
```

```

        PrintWriter prwtr                =        new
PrintWriter(clientskt.getOutputStream(),
true);
    //read from client and send the result.
Server is in a for ever loop
    while ( true)
    {
        // read a line create a
StringTokenizer
        StringTokenizer stkn = new
StringTokenizer(bufrcclient.readLine());
        //conver String to Integer
        id= new
Integer(stkn.nextToken()).intValue();
        System.out.println("\n id received from
the client" + id);
        // find the index number from rolls
array
        for( int i=0;i<rolls.length;i++)
        {
            if( id==rolls[i])
            index=i;}
        // Get the grade*/
        gr=grade[index];
        // send grade to client
        prwtr.println(gr);
        // print on server console too
        System.out.println("Grade of "+ id + "
is :"+ gr);}
    }catch(IOException e){}
    }
}

```

---

## **Example 22.6b: Write Java Client Server Program Wherein Client Submits Roll Number and Gets Grade from Server—Client Side Programming**

```
package com.oops.chap22;
import java.io.*;
import java.util.*;
import java.net.*;
public class ClientDemo2 {
    public static void main(String[]
args)throws IOException {
    int id, index=0;String gr;
    //socket to connect to server
    Socket skt = new
Socket("localhost",5000);
    // Create a BufferedInputStream to get
the data from the server
    BufferedReader bufrserver=new
BufferedReader
    ( new
InputStreamReader(skt.getInputStream()))
;
    //create buffer writer stream for
```

```
    sending data to server
    PrintWriter prwtr = new
    PrintWriter(skt.getOutputStream(),true);
    //continuously send roll number and get
    grade from the server.
    while ( true)
    { Scanner scn= new Scanner(System.in);
      System.out.println("\n Enter roll
number< 50595 to 50599>");
      id = scn.nextInt();
      // send id to server
      prwtr.println(id);
      // get grade from the server
      StringTokenizer stkn =new
      StringTokenizer(bufserver.readLine());
      System.out.println("Grade received from
server of "+ id + " is :"+
      stkn.nextToken());
    }
  }
}
```

---



```
C:\WINDOWS\system32\cmd.exe - java com/oops/chap23/ServerDemo2
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
Could not find the main class: com/oops/chap23/ServerDemo2. Program will exit.

\OopJava\WORKSP\1\oopstech\src>cd..
\OopJava\WORKSP\1\oopstech>cd bin
\OopJava\WORKSP\1\oopstech\bin>java com/oops/chap23/ServerDemo2

d received from the client50595
ade of 50595 is :A

d received from the client50599
ade of 50599 is :A

d received from the client50595
ade of 50595 is :A

d received from the client50596
ade of 50596 is :C
```

```
C:\WINDOWS\system32\cmd.exe - java com/oops/chap23/ClientDemo2

C:\OopJava\WORKSP\1\oopstech\src>cd..
C:\OopJava\WORKSP\1\oopstech>cd bin
C:\OopJava\WORKSP\1\oopstech\bin>java com/oops/chap23/ClientDemo2

Enter roll number< 50595 to 50599>
50595
Grade received from server of 50595 is :A

Enter roll number< 50595 to 50599>
50599
Grade received from server of 50599 is :A

Enter roll number< 50595 to 50599>
50595
Grade received from server of 50595 is :A

Enter roll number< 50595 to 50599>
50596
Grade received from server of 50596 is :C

Enter roll number< 50595 to 50599>
```

## 22.6.2 Multiple Clients and Server Programs Using Multithreads

In real life, there will be several clients wishing to connect to server at the same

tine. For example, when results are declared, several students rush in to connect to server. The best way to handle such a situation is to put each client on a separate thread. The server can achieve the result by

---

```
        ServerSocket servskt = new
ServerSocket (6000);
        while( true) {
            Socket clientskt =
servskt.accept();
            Thread thrd = new
Thread(clientskt);}
```

---

Each loop will create a new connection on server socket. New thread handles the communication between Server and multiple client.

Our next example shows multiple clients being handled by thread. The user will send the server a number and the server returns Boolean true or false to indicate if the number of even or odd.

## Example 22.8a: Write Java Client Server Program—Server to Handle Multiple Clients Using Thread—Server Side Programming

```
package com.oops.chap22;
import java.io.*;
import java.util.*;
import java.net.*;
public class MultipleClientServer {
public static void main(String[]
args)throws IOException {
try {
//server socket
ServerSocket servskt = new ServerSocket
(4000);
int count=0; // counting thread
while ( true)
{
// listen for a request
Socket clientskt = servskt.accept();
System.out.println("\n Starting
thread No "+count);
//create a new thread by calling the
constructor of class CommThread
CommThread thrd = new
CommThread(clientskt,count);
```

```

        thrd.start();
    } //end of while
} catch(IOException e){}
} //end of main
} //end of class
class CommThread extends Thread
{ private Socket clientskt ; //
connected
    private int count; // count for label
the thread
    CommThread(Socket s , int i)
{clientskt=s; count=i;}
    public void run()
{ boolean evenodd=true;
    try
{ // Create a BufferedReadStream to
get the data from the client
BufferedReader bufrclient=new
BufferedReader
    ( new
InputStreamReader(clientskt.getInputStre
am())));
    //create buffer writer stream for
sending data to client
    PrintWriter prwtr = new
PrintWriter(clientskt.getOutputStream
(),true);
    // Serve in a continuous loop
    while ( true)
    {// receive the string data from
client
        StringTokenizer stkn = new

```

```
StringTokenizer(bufrclient.readLine());
    //get year
    int num = new
Integer(stkn.nextToken()).intValue();
    System.out.println("\n Number Received
From Client "+ num);
    //check even or odd
    if ( num%2!=0)evenodd=false;
    else evenodd=true;
    //send evenodd to client
    prwtr.println(evenodd);
    System.out.println("\n Number "+ num
+" is :"+ evenodd);
}
} catch (IOException e ){}
}
```

---

## **Example 22.8b: Write Java Client Server Program for Client–Server to Handle Multiple Clients Using Thread–Client Side Programming**

---

```
package com.oops.chap22;
import java.util.*;
```

```

import java.io.*;
import java.net.*;
public class MultipleclientsClient {
public static void main(String[]
args)throws IOException {
//socket to connect to server
Socket skt = new
Socket("localhost",4000);
// Create a BufferedInputStream to get
the data from the server
BufferedReader bufrserver=new
BufferedReader
( new
InputStreamReader(skt.getInputStream()))
;
//create buffer writer stream for
sending data to server
PrintWriter prwtr = new
PrintWriter(skt.getOutputStream(),true);
//continuously send number and get grade
from the server.
while ( true)
{ Scanner scn= new Scanner(System.in);
System.out.println("\n Enter number<
50595 to 50599>");
int id = scn.nextInt();
// send id to server
prwtr.println(id);
// get even or odd from the server
StringTokenizer stkn =new
StringTokenizer(bufrserver.readLine());
System.out.println("EvenOdd received

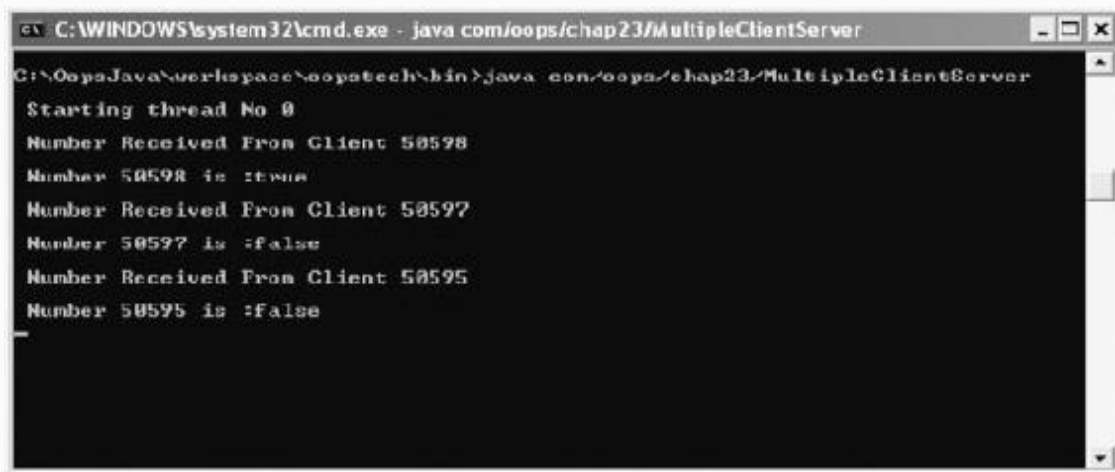
```

```

from server of "+ id + " is :"+
stkn.nextToken());
}}}

```

---



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe - java com/oops/chap23/MultipleClientServer". The window shows the output of the command `java com/oops/chap23/MultipleClientServer`. The output is as follows:

```

Starting thread No 0
Number Received From Client 50598
Number 50598 is :true
Number Received From Client 50597
Number 50597 is :false
Number Received From Client 50595
Number 50595 is :false

```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe - java com/oops/chap23/MultipleClientsClient". The window shows the output of the command `java com/oops/chap23/MultipleClientsClient`. The output is as follows:

```

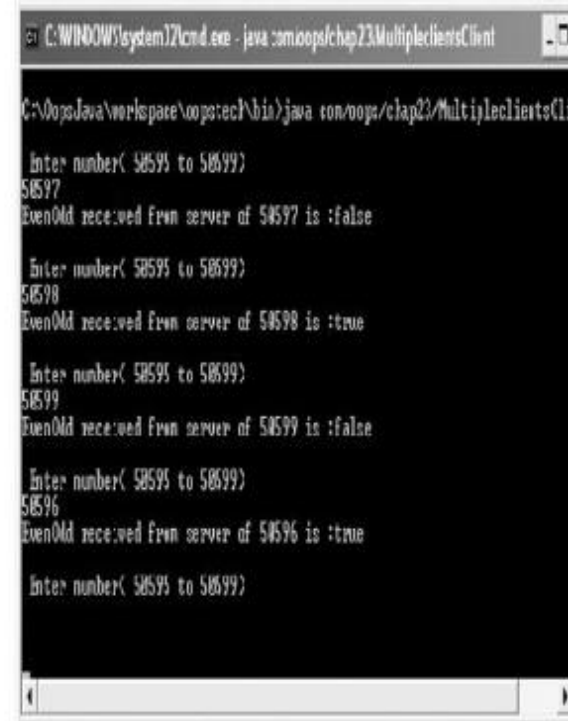
Enter number( 50595 to 50599)
50595
EvenOdd received from server of 50595 is :false

Enter number( 50595 to 50599)
50598
EvenOdd received from server of 50598 is :true

Enter number( 50595 to 50599)
50599
EvenOdd received from server of 50599 is :false

Enter number( 50595 to 50599)

```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe - java com/oops/chap23/MultipleClientsClient". The window shows the output of the command `java com/oops/chap23/MultipleClientsClient`. The output is as follows:

```

Enter number( 50595 to 50599)
50597
EvenOdd received from server of 50597 is :false

Enter number( 50595 to 50599)
50598
EvenOdd received from server of 50598 is :true

Enter number( 50595 to 50599)
50599
EvenOdd received from server of 50599 is :false

Enter number( 50595 to 50599)
50596
EvenOdd received from server of 50596 is :true

Enter number( 50595 to 50599)

```

### 22.6.3 Client Server Program for File Download from Server

Many a times we need to download a file located in a server. The solved Example 4a and 4b shows the technique to download file from a remote server. As usual at server, we need a server socket and a client-specific socket. Then we need `BufferedReader` to accept data from the client and `DataOutputStreamReader` to send the data to client.

## 22.7 Summary

1. `Java.net` is a package that supports networking programs and allows us to create unix-like sockets and implementation of TCP/IP sockets.
2. By attaching those stream files with networking protocols, you can write to files on the Internet and retrieving data from files over the Internet as if you are writing files on to disk.
3. Networking is the ability to connect to Internet from our stand-alone applications or Applets. **Network** is a collection of computers, either homogenous or heterogeneous.
4. **Internet** is a collection or grouping together of networks. We can say that Internet is a network of networks, comprising of millions of computers. These



computers communicate with each other using Internet protocols.

5. **World Wide Web (WWW)** is a way of accessing information on the Internet. It uses HTTP to exchange data amongst communicating computers.
6. Computers running on the Internet communicate with each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).
7. A protocol is a set of rules to be followed by cooperating systems on the Internet for transferring data from one system to another system.
8. TCP/IP, i.e., Transmission Control Protocol and Internet Protocol, are standard protocols to be used for transmission of data.
9. TCP/IP is a connection-oriented and reliable service.
10. Connection oriented means the sender confirms if the receiver has indeed received the data or not. If not received then retransmission of data takes place.
11. TCP/IP is a point-to-point reliable communication, i.e., it is an acknowledged service that ensures correct reception of data at receiver.
12. TCP/IP has five layers, viz., Application Layer, Transportation Layer, IP Layer, Data Link Layer and Physical Layer.
13. Address of destination can be specified by numeric address like 192.127.68.24 or by domain name like [www.google.com](http://www.google.com).
14. Domain name server transforms address from numeric to web site name and vice versa.
15. Data Link Layer receives the Frame from IP layer and dispatches the frame to destination computer.
16. Network card maps the IP address into 48-bit unique Network card identification number and transmits based on MAC protocol.
17. MAC protocol simply tells which is going on the Net next, i.e., it tells the rules for sharing single transmission

- medium by multiple frames.
18. UDP is a connection-less service. This means that communication is not guaranteed between two applications. Delivery is not important and is not guaranteed. It is mostly used in transmission of video and audio, etc.
  19. We can obtain Internet address for any web site by using `InetAddress` class of `java.net` package. `Inet` class throws `UnknownHostException`.
  20. `URLConnection` is a class that allows us to examine the properties prior to transporting.
  21. `Socket` has unique identification number also called port number and takes 16 bits (2 bytes).
  22. `ServerSocket` created using `ServerSocket` class and second is client side socket, called `socket` created using a `socket` class.

## Exercise Questions

### Objective Questions

1. Networking classes are included in

1. `java.IO.*`;
2. `java.net.*`;
3. `java.util.*`;
4. `java.lang.*`;

2. URL stands for

1. Universal Resource Locator
2. Unified Resource Locator
3. Uniform Resource Locator
4. Unique Resource Locator

3. Which of the following classes contain IP Address?

1. Inet class
2. URL class
3. URL Connection class
4. Socket class

4. Which of the following statements are true in respect of TCP/IP?

1. Connection oriented means acknowledgement from receiver
2. TCP/IP is a point-to-point service
3. There is a separate network layer in TCP/IP
4. TCP/IP is a reliable service means acknowledgement else retransmission

1. i, ii and iv
2. i, ii and iii
3. ii, iii and iv
4. i and iii

5. Output of transport layer of TCP/IP is

1. continuous stream of bytes
2. frame
3. packet
4. MAC Frame

6. Output of Application layer of TCP/IP is

1. continuous stream of bytes
2. frame
3. packet
4. MAC Frame

7. Port Numbers are represented by

1. 8 bits
2. 16 bits
3. 32 bits
4. 48 bits

8. Port Number for http protocol is

1. 110
2. 60
3. 80
4. 22

9. Port Number for post office protocol (POP) & FTP are

1. 25 and 119
2. 53 and 80
3. 110 and 25
4. 110 and 21

10. Output of IP layer of TCP/IP is

1. continuous stream of bytes
2. frame
3. packet
4. MAC Frame

11. `accept()` method returns

1. Client socket
2. Server Socket
3. Socket
4. Datagram socket

#### **Short-answer Questions**

12. What is a socket?
13. What is a port number?
14. What is an IP Address?
15. When IP Address is available what is the need of port address also?
16. Explain the role of `ServerSocket`.
17. Explain the role of socket class for client.
18. Distinguish TCP/IP connection and UDP connection service.
19. Explain how a server listens for a connection at a port.

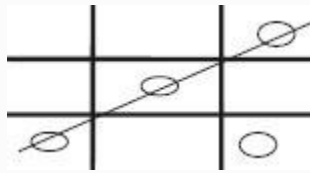
#### **Long-answer Questions**

20. Explain the function of each layer of TCP/IP Layers.
21. Explain the process of creation of `ServerSocket` and socket for client with suitable examples.
22. Explain the steps involved in setting up client server networking.
23. Explain the role of Stream Readers and Writers and Buffered Stream in networking programs.

24. How are multiple clients handled by a single server?
25. Explain how a line of text can be obtained from the server.
26. Explain how primitive data can be exchanged between server and a client.

#### Assignment Questions

27. Write a client and server program wherein a student enters the roll number and server houses random file that holds student results. Obtain the result and display on the client console.
28. Write a program for Tic Tac toe server and client. Tic tac game is the game you would have played at some time or the other. Hint: Assume two-dimensional matrix display at both server and client and use two-way communication model.



29. Rewrite Ex 1 for multiple client paradigm.
30. Modify two-way chat program to handle conference mode wherein there are three peer systems and messages are displayed at all terminals as on broadcast.
31. Modify the problem at 4 to cater to sending messages only to intended clients.
32. Write a client server program wherein `BufferedReader` is used for reading in the data and `PrintStream` is used for sending data.

#### Solutions to Objective Questions

1. b

- 2. c
- 3. a
- 4. a
- 5. c
- 6. a
- 7. b
- 8. c
- 9. d
- 10. b
- 11. a

# 23

## Graphics Using Swing Components and Applets

### LEARNING OBJECTIVES

*At the end of this chapter, you will be able to understand*

- Use of Abstract Window Tool (AWT) kit.
- Hierarchy of Graphics Software and their components and their usage.
- Event types and event listeners.
- Applets and their usage with swing components.
- Converting Applet into Application and vice versa.

### 23.1 Graphics Programming and Applets Introduction

The programs we have seen till now had text-based output. They were uninspiring and insipid. Today's world is graphics and client server programs with audio–video integration programs. Moreover these applications are all Internet-enabled and web-based applications. Web applications use extensively graphical user interfaces (GUI). This chapter introduces you to these GUI features like frames, panels, text boxes, buttons, menus, etc. These are collectively provided in a toolkit called Advanced Windowing Tool (AWT) Kit. AWT classes provide platform-independent interfaces and their implementations. This chapter introduces you to the concept of event-driven programs, frames and panels which are extensively used in application development.

Java has been designed to be a networking language. It can claim so because it supports Applet. Applets are small java programs compiled into byte codes and the class can be transported anywhere in the network using HTML (Hyper Text Mark Up Language) Tags. The system receiving this

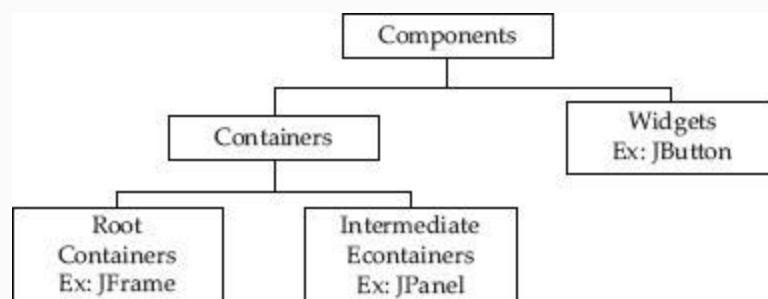


class runs the class file imported using java virtual machine and executes the applet code on a local machine.

Finally java has introduced swing components with better feel and look than applets. This chapter handles all interfaces of AWT using Swing components only.

## 23.2 Hierarchy of Graphics Software

Graphical User Interface (GUI) development in JAVA is based on components. Java components can further be divided into containers and widgets, as shown in Figure 23.1.



**Figure 23.1** Java GUI hierarchy

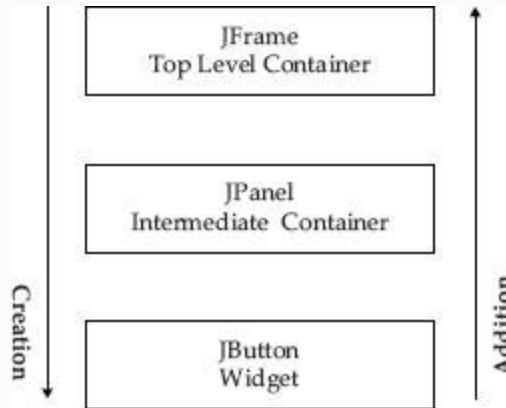
Containers are further classified as Root Containers and Intermediate Containers. Root Containers, also called top-level containers, are necessary for every GUI program. Swing provides developers with three top-level containers: JFrame, JDialog and JApplet. Intermediate containers are not mandatory but are used widely as they help in managing and organizing widgets. JPanel is the most widely used intermediate container in Swing.

Widgets are GUI elements which are used to interact with the user; a typical use of widgets is to display content to user in the form of text, images, menus, buttons, checkboxes and radio buttons. Widgets are also used to get user inputs. JButton, JLabel, JScrollBar and JCheckBox are examples of widgets in Swing.

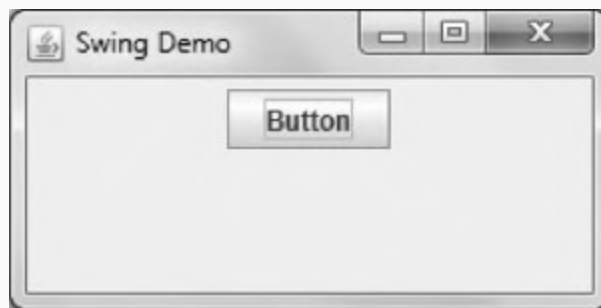
### 23.3 Containers in Swing

Every Java GUI application should be designed with a containment hierarchy, with the root of the hierarchy being one of the Root containers, i.e., JFrame, JDialog or

JApplet. For Swing Applications the root is always JFrame. Figure 23.3 shows details.



**Figure 23.2** Swing containment hierarchy



**Figure 23.3** Button

Figure 23.2 shows the containment hierarchy for a program that has just one

JButton widget. In the figure, the intermediate container JPanel is optional, but the top-level container is compulsory. The figure also depicts the creation and addition of rules; for java GUI programs the creation is from top to bottom, which means that we create the JFrame component first and the JButton component last, while addition starts from bottom. Example 23.1 illustrates the creation and addition rules for Java GUI programs.

### Example 23.1: Creating JFrame and Adding GUI Components

```
1. import javax.swing.*;
2. public class SwingDemo {
3.     public static void main(String[]
args) {
4.         JFrame frame = new JFrame("Swing
Demo");
5.         JPanel panel = new JPanel();
6.         JButton button = new
JButton("Button");
```

```
7.  
8. panel.add(button);  
9. frame.setContentPane(panel);  
10. frame.setVisible(true);  
11. }  
12. }
```

---

## Output

As seen from the example above, we instantiate the root container frame first in line 4; we then instantiate the intermediate container panel in line 5 and finally a button widget in line 6. That is, we follow the top-down approach during creation. But during addition, we add the button to panel first in line 7 then we add the panel to the frame in line 8 and finally display the frame in line 9, thus following the bottom-up approach.

The addition and creation rules will apply to all programs discussed in this chapter.

### *23.3.1 JFrame Class*

A frame typically provides many services; some of the more common services include ability to set the title, set icon image, provide menu bar services, set height and width of

the frame, and set location of the frame. To manage the setting of the frame and to make the code reusable, we create a separate class for the frame called `DisplayFrame`. In `DisplayFrame`'s constructor we define all the default settings for our root frame. Figure 23.3 shows the details.

### **Example 23.2: Program to Demonstrate `JFrame` Class**

```
1. import javax.swing.*;
2. public class SwingDemo {
3.     public static void main(String[]
args) {
4.         JFrame frame = new
DisplayFrame();
5.         frame.setVisible(true);
6.     }
7. }
8. class DisplayFrame extends JFrame{
9.     DisplayFrame() {
10.
setDefaultCloseOperation(JFrame.EXIT_ON_
```

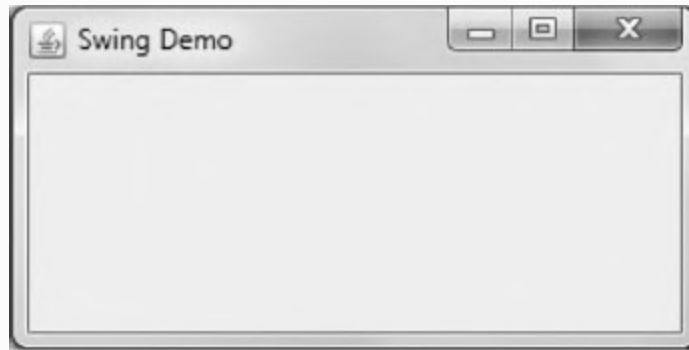
```
CLOSE);  
    11.    setTitle("Swing Demo");  
    12.    setSize(300, 150);  
    13.    setLocation(100,100);  
    14. }  
    15. }
```

---

The `JFrame` class is present in `javax.swing.JFrame`. To use this class, we use the import statement `import javax.swing.*`, as shown in line 1. In line 4, we instantiate the `DisplayFrame` object, which in turn invokes the constructor `DisplayFrame ( )` in line 9. Inside the constructor we set some of the settings for our root frame.

The method `setDefaultCloseOperation ( )` in line 10 with the parameter `JFrame.EXIT_ON_CLOSE` causes the frame to close whenever the user closes the frame using the close icon. The method `setTitle ( )` in line 11 sets the frame name. The method `setSize ( )` in line 12 sets the width and height for the frame and

finally the method `setLocation ( )` in line 13 sets the x, y location on the screen. Output is in Figure 23.4.



**Figure 23.4** A frame in java swing

### 23.3.2 *JPanel Class*

Panels are intermediate containers and as such not essential for creating Java graphical user interface programs. But panels offer a lot of flexibility in managing widgets so it is always advisable to have panels in our programs. Similar to the `DisplayFrame` class which extends the basic `JFrame` class, we create a new class



called `DisplayPanel` which extends the `JPanel` class.

### **Example 23.3: Program to Demonstrate `JPanel` Class**

```
1. import javax.swing.*;
2. public class SwingDemo {
3.     public static void main(String[]
args) {
4.         JFrame frame = new
DisplayFrame();
5.         frame.setVisible(true);
6.     }
7. }
8. class DisplayFrame extends JFrame{
9.     DisplayFrame() {
10.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
11.         setTitle("Swing Demo");
12.         setSize(300, 150);
13.         setLocation(100,100);
14.         JPanel panel = new
DisplayPanel();
15.         getContentPane().add(panel);
```

```
16. }  
17. }  
18. class DisplayPanel extends  
JPanel{  
19.     DisplayPanel() {  
20.         //Widgets will be added here  
21.     }  
22. }
```

---

The code of [Example 23.3](#) is almost similar to [Example 23.2](#). In the above example, we instantiate the `DisplayPanel` object in line 14. The constructor for `DisplayPanel` is presently empty but going forward, as we will see in subsequent examples, all the widget creation code will be placed inside this class. Each `JFrame` object has a contentpane to which all the components you want to display have to be added. Line 15 demonstrates how to get the contentpane for your frame and add the panel.

## 23.4 Display Widgets in Swing

Widgets are the components which help us build the user interfaces. They are the components that the user actually sees and

interacts with users. Typical examples include checkboxes, buttons and menus. Java Swing provides many widgets; in subsequent sections, we discuss some of the most frequently used widgets.

### *23.4.1 JLabel Class*

The JLabel object is used to display text, images or both text and images in Java.

Figure 23.5 shows a typical use case scenario, wherein the string “Welcome to Java” is displayed using a JLabel object. Example 23.4 demonstrates how to create and display JLabel objects in Java.



**Figure 23.5** JLabel

## Example 23.4: Program to Demonstrate JLabel Class

```
1. import javax.swing.*;
2. public class SwingDemo {
3.     public static void main(String[]
args) {
4.         JFrame frame = new
DisplayFrame();
5.         frame.setVisible(true);
6.     }
7. }
8. class DisplayFrame extends JFrame{
9.     DisplayFrame(){
10.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
11.         setTitle("Swing Demo");
12.         setSize(300, 150);
13.         setLocation(100,100);
14.         JPanel panel = new
DisplayPanel();
15.         getContentPane().add(panel);
16.     }
17. }
18. c DisplayPanel extends JPanel{
19. DisplayPanel() {
```

```
20. JLabel label = new JLabel
21. ("Welcome to Java Swing" );
22. add(label);
23. }
24. }
```

---

The `JLabel` class is present in `javax.swing.JLabel`. To use this class, we use the import statement `import javax.swing.*`, as shown in line 1. We instantiate the `JLabel` object in line 20; the string to be displayed is passed as a parameter during instantiation. Finally, we add the `JLabel` object `label` to the panel in line 22.

### *23.4.2 JButton*

The `JButton` object is used to display buttons in Java. Buttons are very useful user interface elements, which can be used for a variety of user interactions. [Figure 23.6](#) shows a typical use case scenario, wherein the button "Click Me" is displayed using a `JButton` object. [Example 23.5](#) demonstrates how to create and display `JButton` objects in Java.



**Figure 23.6** JButton

## **Example 23.5: Program to Demonstrate JButton Class**

```
1. import javax.swing.*;
2. public class SwingDemo {
3.     public static void main(String[]
args) {
4.         JFrame frame = new
DisplayFrame();
5.         frame.setVisible(true);
6.     }
7. }
8. class DisplayFrame extends JFrame{
9.     DisplayFrame(){
```

```

10.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
11.    setTitle("Swing Demo");
12.    setSize(300, 150);
13.    setLocation(100,100);
14.    JPanel panel = new
DisplayPanel();
15.    getContentPane().add(panel);
16. }
17. }
18. class DisplayPanel extends
JPanel{
19. DisplayPanel() {
20. JButton button = new
JButton("Click Me");
21. add(button);
22. }
23. }

```

---

The JButton class is present in `javax.swing.JButton`. To use this class, we use the import statement `import javax.swing.*`, as shown in line 1. We instantiate the JButton object in line 20.

The string to be displayed is passed as a parameter during instantiation. Finally, we

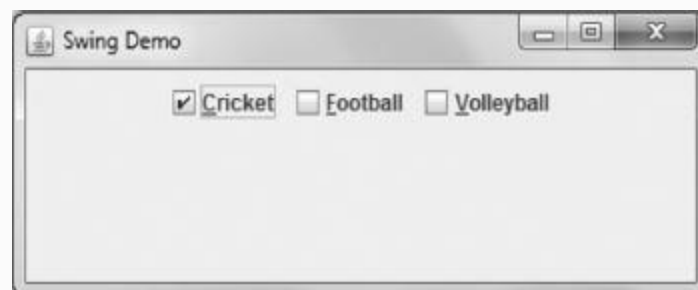
add the JButton object button to the panel in line 21.

### 23.4.3 *JCheckBox*

The JCheckBox object is used to display checkboxes in Java. Checkboxes allow users to either select or deselect an item; they also allow multiple items to be selected at the same time. The figure below shows a typical use case scenario, wherein three checkboxes “Cricket”, “Football” and “Volleyball” are displayed using a JCheckBox object.

Example 23.6 demonstrates how to create and display JCheckBox objects in Java.

Figure 23.7 shows the output.



**Figure 23.7** JCheckbox

---



## Example 23.6: Program to Demonstrate JCheckBox Class

```
1. import java.awt.event.KeyEvent;
2. import javax.swing.*;
3. public class SwingDemo {
4.     public static void main(String[]
args) {
5.         JFrame frame = new
DisplayFrame();
6.         frame.setVisible(true);
7.     }
8. }
9. class DisplayFrame extends JFrame{
10.     DisplayFrame(){
11.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
12.         setTitle("Swing Demo");
13.         setSize(400, 150);
14.         setLocation(100,100);
15.         JPanel panel = new
DisplayPanel();
16.         getContentPane().add(panel);
17.     }
18. }
19. class DisplayPanel extends
```

```
JPanel{
    20.  DisplayPanel() {
    21.      JCheckBox cb1 = new
JCheckBox("Cricket");
    22.
cb1.setMnemonic(KeyEvent.VK_C);
    23.      cb1.setSelected(false);
    24.      JCheckBox cb2 = new
JCheckBox("Football");
    25.      cb2.setMnemonic(KeyEvent.VK_F);
    26.      cb2.setSelected(false);
    27.      JCheckBox cb3 = new
JCheckBox("Volleyball");
    28.      cb3.setMnemonic(KeyEvent.VK_V);
    29.      cb3.setSelected(false);
    30.      add(cb1);
    31.          add(cb2);
    32.          add(cb3);
    33.  }
    34. }
```

---

The `JCheckBox` class is present in `javax.swing.JCheckBox`. To use this class, we use the import statement `import javax.swing.*`, as shown in line 2. We instantiate the `JCheckBox` object in line 21; the string to be displayed is passed as a parameter during instantiation. The method

`setMnemonic()` allows users to select or deselect items in checkboxes using keyboard keys. The statement in line 22 uses the `setMnemonic` method, thereby allowing users to select or deselect the Cricket checkbox using ALT + C button combination on the keyboard. The `setMnemonic()` function takes a `KeyEvent` as the input parameter; `KeyEvents` are represented using the syntax `VK_*`, for example, C is represented as `VK_C`. `KeyEvents` are defined in *java.awt.event.KeyEvent*. To use `KeyEvents` in our program, we use the import statement `import java.awt.event.KeyEvent`, as shown in line 1. `setSelected()` method as seen in line 23 is used to set the initial state for a checkbox; `false` indicates that the checkbox is initially deselected. We follow the same procedure of instantiating a checkbox and setting its properties for other checkboxes as well. Finally, we add the `JCheckBox` objects `cb1`, `cb2` and `cb3` to the panel in lines 30, 31, and 32.

### 23.4.4 JComboBox

The JComboBox object is used to display checkboxes in Java. Combo boxes in Java have a button and a drop-down list. The button is used to activate the drop-down list. Combo boxes offer the advantage of space and at the same time provide functionality of letting the user choose a particular item from a list of items. The attached figure shows a typical use case scenario, wherein on pressing the down arrow button of the Combo box, a drop-down list containing names of various cities is displayed to the user using a JComboBox object. Example 23.7 demonstrates how to create and display JComboBox objects in Java. Figure 23.8 shows the output.



**Figure 23.8** JCombo box

## **Example 23.7: Program to Demonstrate JComboBox Class**

```
1. import javax.swing.*;
2. public class SwingDemo {
3.     public static void main(String[]
args) {
4.         JFrame frame = new
DisplayFrame();
5.         frame.setVisible(true);
6.     }
7. }
8. class DisplayFrame extends JFrame{
9.     DisplayFrame() {
```

```

10.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
11.     setTitle("Swing Demo");
12.     setSize(300, 150);
13.     setLocation(100,100);
14.     JPanel panel = new
DisplayPanel();
15.     getContentPane().add(panel);
16. }
17. }
18. class DisplayPanel extends JPa
nel{
19. DisplayPanel() {
20. String[] cities = { "Delhi",
"London", "New York", "Moscow", "Mumbai"
};
21.     JComboBox citylist = new
JComboBox(cities);
22.     citylist.setSelectedIndex(2);
23.     add(citylist);
24. }
25. }

```

---

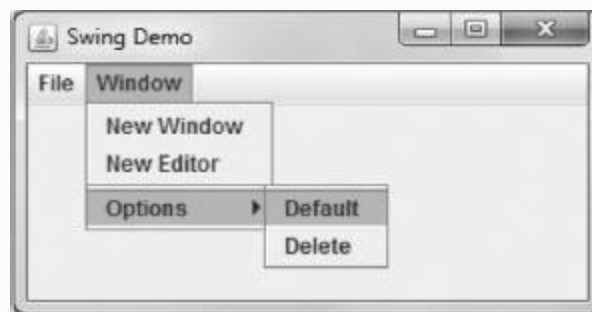
The JComboBox class is present in javax.swing.JComboBox. To use this class, we use the import statement import javax.swing.\*, as shown in line 1. In line 20, we define cities, an array of Strings, which

contains the names of all the items we would like to display in our drop-down list. We instantiate the JComboBox object in line 21; the array cities is passed as a parameter during instantiation. The `setSelectedIndex()` method in line 22 shows how to set the default display item for the combo box. We have passed 2 as the parameter to `setSelectedIndex()`, so when the combo box is instantiated, the default item displayed will be “New York” as the String New York is present at index 2 in the array cities. Finally, we add the JComboBox object citylist to the panel in line 23.

#### *23.4.5 JMenu Class*

Menus in Java are implemented using three components, namely, JMenuBar, JMenu and JMenuItem. Menu bar is generally located at the top of the window; it contains one or more menus. When we click on the menu, a list appears which contains user selectable menu items or in some cases a submenu. The figure below shows a typical

use case scenario, wherein the menu bar has two menus, File and Window. The menu Window has two menu items, New Window and New Editor, and a submenu called Options. Example 23.8 demonstrates how to create and display complete menus using JMenuBar, JMenu and JMenuItem objects in Java. Output is shown in Figure 23.9.



**Figure 23.9** JMenu

## **Example 23.8: Program to Demonstrate JMenu Class**



---

```
1. import javax.swing.*;
2. public class SwingDemo {
3.     public static void main(String[]
args) {
4.         JFrame frame = new
DisplayFrame();
5.         frame.setVisible(true);
6.     }
7. }
8. class DisplayFrame extends JFrame{
9.     DisplayFrame(){
10.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
11.         setTitle("Swing Demo");
12.         setSize(300, 150);
13.         setLocation(100,100);
14.         JMenuBar MenuBar = new
JMenuBar();
15.         JMenu menu = CreateFileMenu();
16.         MenuBar.add(menu);
17.         menu = CreateWindowMenu();
18.         MenuBar.add(menu);
19.         setJMenuBar(MenuBar);
20.     }
21.     private JMenu CreateFileMenu() {
22.         JMenu menu = new JMenu("File");
23.         JMenuItem item = new
JMenuItem("Open");
24.         menu.add(item);
25.         item = new JMenuItem("Close");
```

```
26.    menu.add(item);
27.    menu.addSeparator();
28.    item = new JMenuItem("Exit");
29.    menu.add(item);
30.    return menu;
31. }
32. private JMenu CreateWindowMenu()
{
    33.    JMenu menu = new
JMenu("Window");
    34.    JMenuItem item = new
JMenuItem("New Window");
    35.    menu.add(item);
    36.    item = new JMenuItem("New
Editor");
    37.    menu.add(item);
    38.    menu.addSeparator();
    39.    JMenu submenu = new
JMenu("Options");
    40.    item = new
JMenuItem("Default");
    41.    submenu.add(item);
    42.    item = new JMenuItem("Delete");
    43.    submenu.add(item);
    44.    menu.add(submenu);
    45.    return menu;
    46. }
    47. }
```

---

The JMenuBar, JMenu and JMenuItem classes are present in javax.swing.JMenuBar, javax.swing.JMenu and javax.swing.JMenuItem, respectively. To use these classes, we use the import statement `import javax.swing.*`, as shown in line 1. We define two functions `CreateFileMenu( )` in line 21 and `CreateWindowMenu( )` in line 32. The function `CreateFileMenu( )` instantiates a menu with the name “File” in line 22; it also creates two menu items “Close” and “Exit” in lines 25 and 28 and adds these menu items to the File menu in lines 26 and 29. The `addSeparator( )` method adds a line between the two menu items in File Menu. The `CreateWindowMenu( )` function creates a menu “Window” and populates it with two menu items “New Editor” and “New Window”. The function also creates a submenu “Options” in line 39, adds menu items “Default” and “Delete” and then adds the submenu to the Window menu in line 44.

Menu bar is instantiated in line 14, the menus “File” and “Windows” are instantiated in lines 15 and 17 and added to the menu bar in lines 16 and 18. The menu bar differs from other widgets discussed till now as a menu bar is added to the Frame rather than to an intermediate panel. The method `setJMenuBar ( )` is used to add the menu bar to the frame, as shown in line 19.

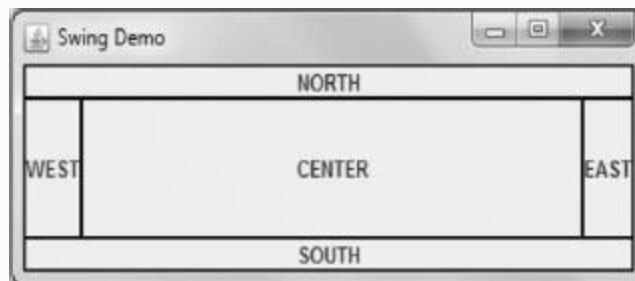
## 23.5 Layout Managers

Till now we have discussed about creation of widgets; most of our examples so far have single widgets. But in real-life scenarios, we will have multiple widgets on the same screen and we need some kind of layout management to manage the placement of these widgets on the screen. Java Swing provides us with three Layout managers to simplify our task. In subsequent sections, we will have a closer look at these layout managers.

### *23.5.1 Border Layout*

Border layout divides the panel into five regions: North, South, East, West and Center, as shown below. In Border Layout, the widgets present in the North, South and Center expand to fill the available width whereas the widgets in the East and West regions expand as per the widget contents.

Example 23.9 illustrates how to arrange widgets in Border Layout. Figure 23.10 shows the output.



**Figure 23.10** Border layout

### **Example 23.9: Program to Demonstrate Border Layout**

---

```
1. import java.awt.*;
2. import javax.swing.*;
3. import javax.swing.border.Border;
4. public class SwingDemo {
5.     public static void main(String[]
args) {
6.         JFrame frame = new
DisplayFrame();
7.         frame.setVisible(true);
8.     }
9. }
10. class DisplayFrame extends
JFrame{
11.     DisplayFrame() {
12.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
13.         setTitle("Swing Demo");
14.         setSize(400, 150);
15.         setLocation(100,100);
16.         JPanel panel = new
DisplayPanel();
17.         getContentPane().add(panel);
18.     }
19. }
20. class DisplayPanel extends
JPanel{
21.     DisplayPanel() {
22.         setLayout(new BorderLayout());
23.         add(new JLabel("NORTH"),
BorderLayout.NORTH);
```

```

24.      add(new newLabel("SOUTH"),
BorderLayout.SOUTH);
25.      add(new newLabel("EAST"),
BorderLayout.EAST);
26.      add(new newLabel("WEST"),
BorderLayout.WEST);
27.      add(new
newLabel("CENTER"), BorderLayout.CENTER);
28.  }
29.  class newLabel extends JLabel {
30.      Border black =
BorderFactory.createLineBorder(Color.black);
31.      newLabel(String S){
32.          setText(S);
33.
setHorizontalAlignment(CENTER);
34.          setBorder(black);
35.      }
36.  }
37.  }

```

---

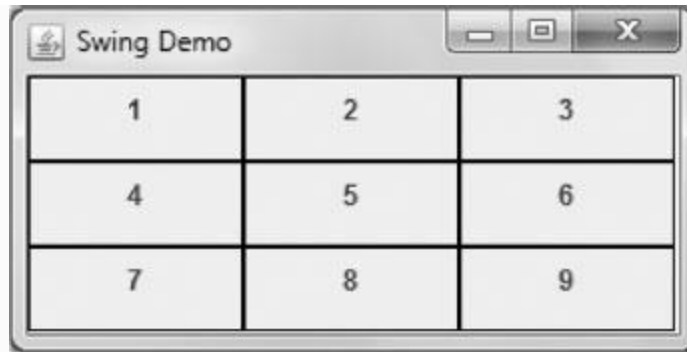
The panel layout is set using the method *setLayout()* and passing an instance of the class `BorderLayout`, as seen in line 22. We use the *add()* method to populate the panel with `newLabel` widgets along with the location information, as seen in lines 23 to 27. The `newLabel` widget is an extension of

JLabel with additional properties. It sets the alignment of text in the label so that the text appears centered; this is achieved using the method *setHorizontalAlignment( )* and passing the parameter CENTER, as seen in line 33. We also set a black border for the label so that the area occupied by the label is clearly visible. This is achieved using the method *setBorder( )* and passing the parameter black, as seen in line 34. The parameter black is of the type Border which is created in line 30. You can create different line colours by changing the parameter in *createLineBorder( )*.

### 23.5.2 Grid Layout

Grid Layout divides the screen into specified rows and columns as seen in the figure shown below. Grid layout is useful when we want to create UI similar to windows desktop wherein the icon is symmetrically placed in the screen. Figure 23.11 shows the output.





**Figure 23.11** GridLayout

## **Example 23.10: Program to Demonstrate Grid Layout**

```
1. import java.awt.*;
2. import javax.swing.*;
3. import javax.swing.border.Border;
4. public class SwingDemo {
5.     public static void main(String[]
args) {
6.         JFrame frame = new
DisplayFrame();
7.         frame.setVisible(true);
8.     }
9. }
```

Figure 23.11 GridLayout

```

10. class DisplayFrame extends
JFrame{
    11.  DisplayFrame() {
    12.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
    13.      setTitle("Swing Demo");
    14.      setSize(300, 150);
    15.      setLocation(100,100);
    16.      JPanel panel = new
DisplayPanel();
    17.      getContentPane().add(panel);
    18.  }
    19. }
    20. class DisplayPanel extends
JPanel{
    21.  DisplayPanel() {
    22.      setLayout(new GridLayout(3, 3));
    23.      for(int i = 1; i <= 9 ; i++ ){
    24.          add(new
CalcLabel(Integer.toString(i)));
    25.      }
    26. }
    27. class CalcLabel extends JLabel {
    28.  CalcLabel(String S){
    29.      Border black =
BorderFactory.createLineBorder(Color.BLA
CK);
    30.      setText(S);
    31.
setHorizontalAlignment(CENTER);
    32.      setBorder(black);

```

```
33.     }  
34.     }  
35. }
```

---

Similar to [Example 23.9](#), the only difference is in the `GridLayout` instantiation, as seen in line 22. We need to specify the number of rows and columns in our present example; the number of rows and columns are three.

### 23.5.3 Flow Layout

Flow layout is the default layout. If you do not specify any layout using the `setLayout()` method, then by default, the panel will have a Flow layout. In flow layout, the widgets are arranged from left to right and then from top to bottom. Flow layout can be seen in [Figure 23.7](#) wherein all the checkboxes are arranged from left to right.

## 23.6 Event Types and Event Listeners

GUI programming in Java involves two parts. The first part involves creating user interface using various components, containers and widgets, whereas the second

part involves handling events that are generated due to the user interaction. Events are generated due to user activity and they can range from key presses, mouse clicks to selecting/deselecting checkboxes and button clicks. The Java Abstract windowing Toolkit (AWT) framework is responsible for delivering user-generated events to the applications. AWT also defines the interfaces for all the event handlers.

Events in Java can be classified into two types: high-level events and low-level events. High-level events such as Action events are specific to certain widgets and are sent to the application only if those widgets are present and have registered for the event. In contrast low-level events such as Keyboard and Mouse events are sent to all the widgets which have registered to receive them. In the subsequent sections, we discuss some of the most commonly encountered events and their handling mechanisms.

### *23.6.1 Action Event Listener*

Action events are one of the most common events in Java. They are generated whenever the user performs any action on the widget like pressing a button or choosing an item from the menu. To enable us to handle action events, we need to create a class that implements the ActionListener interface. The ActionListener interface requires us to implement just one method

*actionPerformed(ActionEvent e).*

The function *actionPerformed()* is invoked whenever the user performs an action on the widget. Example 23.12 illustrates how to create an action event handler and handle action events.

### **Example 23.11: Program to Demonstrate Action Event Handling**

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. public class SwingDemo {
```

```

5.    public static void main(String[]
args) {
6.        JFrame frame = new
DisplayFrame();
7.        frame.setVisible(true);
8.    }
9. }
10. class DisplayFrame extends
JFrame{
11.    DisplayFrame() {
12.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
13.        setTitle("Swing Demo");
14.        setSize(300, 150);
15.        setLocation(100,100);
16.        JPanel panel = new
DisplayPanel();
17.        getContentPane().add(panel);
18.    }
19. }
20. class DisplayPanel extends
JPanel{
21.    int count;
22.    JLabel label;
23.    JButton button;
24.    DisplayPanel() {
25.        count = 0;
26.        button = new JButton("Click
Me");
27.        label = new JLabel();
28.        label.setText("Count = "

```

```

+count);
    29.      button.addActionListener(new
HandleEvent());
    30.      add(button);
    31.      add(label);
    32.  }
    33.  class HandleEvent implements
ActionListener{
    34.      public void
actionPerformed(ActionEvent e) {
    35.          count ++;
    36.          label.setText("Count = "
+count);
    37.      }
    38.  }
    39.  }

```

---

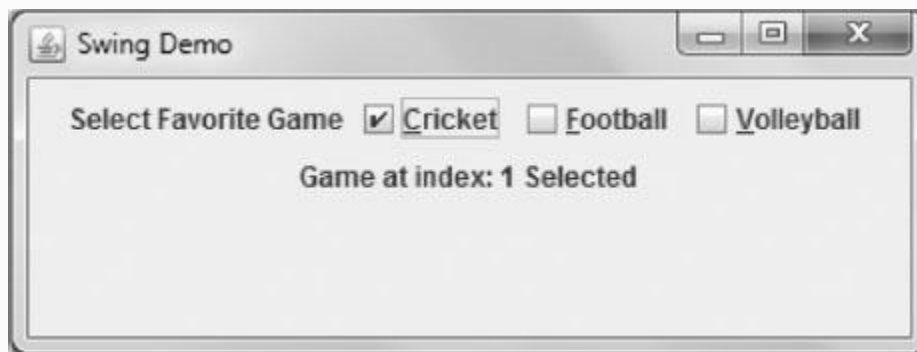
The program above is used to count the number of times the user has pressed the button “Click Me”. The count is updated on the screen, as shown in [Figure 23.13](#). At the start of the program, the count is 0 after 12 clicks by the user the count shows 12. Updating the count on button press is done by the class `HandleEvent` line 33 which implements the `ActionListener` interface. As discussed above, any class implementing the `ActionListener` should implement the

method *actionPerformed()*. This is done in line 34. The function *actionPerformed()* is invoked whenever the user presses the button “Click Me”. In this function, we update the count variable in line 35 and update the label which displays the count using *JLabels setText()* function call in line 36. We have implemented the *HandleEvent* class as an inner class to *DisplayPanel* as it needs to use the variables *count* and *label* defined in *DisplayPanel* class. Line 29 *button.addActionListener(new HandleEvent())* shows how to bind the action event handler with the button widget. Binding widget with event handler as done in line 29 is absolutely essential else the events will not be sent to the event handler.





**Figure 23.12** Output of Example 23.11



**Figure 23.13** Output of Example 23.12

### *23.6.2 Item Event Listener*

Item events are generally fired by components such as checkboxes, radio buttons and combo boxes wherein the user selects or deselects an item. To enable us to handle item events in our program, we need

to create a class that implements the `ItemListener` interface.

The `ItemListener` interface requires us to implement one method

*itemStateChanged (ItemEvent e).*

The function *itemStateChanged ()* is invoked whenever the user selects or deselects an item in the widget. Example 23.12 illustrates how to create an item event handler and handle item events. Figure 23.13 shows the output.

### **Example 23.12: Program to Demonstrate Item Event Handling**

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. public class SwingDemo {
5.     public static void main(String[]
args) {
6.         JFrame frame = new
DisplayFrame();
7.         frame.setVisible(true);
```

```

8.    }
9. }
10.  class DisplayFrame extends
JFrame{
11.    DisplayFrame() {
12.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
13.        setTitle("Swing Demo");
14.        setSize(400, 150);
15.        setLocation(100,100);
16.        JPanel panel = new
DisplayPanel();
17.        getContentPane().add(panel);
18.    }
19. }
20.  class DisplayPanel extends
JPanel{
21.    JLabel label;
22.    JCheckBox cb1;
23.    JCheckBox cb2;
24.    JCheckBox cb3;
25.    DisplayPanel() {
26.        cb1 = new
JCheckBox("Cricket");
27.
cb1.setMnemonic(KeyEvent.VK_C);
28.        cb1.setSelected(false);
29.        cb2 = new
JCheckBox("Football");
30.
cb2.setMnemonic(KeyEvent.VK_F);

```

```

31.      cb2.setSelected(false);
32.      cb3 = new
JCheckBox("Volleyball");
33.
cb3.setMnemonic(KeyEvent.VK_V);
34.      cb3.setSelected(false);
35.      JLabel display_label = new
JLabel();
36.      display_label.setText("Select
Favorite Game");
37.      label = new JLabel();
38.      label.setText("");
39.      HandleItemEvent handler = new
HandleItemEvent();
40.      cb1.addItemListener(handler);
41.      cb2.addItemListener(handler);
42.      cb3.addItemListener(handler);
43.      add(display_label);
44.      add(cb1);
45.      add(cb2);
46.      add(cb3);
47.      add(label);
48.
49.  }
50.  class HandleItemEvent implements
ItemListener{
51.      public void
itemStateChanged(ItemEvent e) {
52.          int index = 0;
53.          Object source =
e.getItemSelectable();
54.          if (source == cb1) {

```

```
55.         index = 1;
56.     } else if (source == cb2) {
57.         index = 2;
58.     } else if (source == cb3) {
59.         index = 3;
60.     }
61.     if (e.getStateChange() ==
ItemEvent.DESELECTED) {
62.         label.setText("Game at index:
" +index + " Deselected");
63.     }
64.     if (e.getStateChange() ==
ItemEvent.SELECTED) {
65.         label.setText("Game at index:
" +index + " Selected");
66.     }
67. }
68. }
69. }
```

---

The program above is designed to get the users favourite sport. As it is a checkbox, multiple selections are possible. The label below the checkboxes records the index of the latest selection; index 1 implies cricket. The label is dynamically updated each time the user selects or deselects a sport. Updating the label is done in the class `HandleItemEvent` defined in line 50 which

implements the `ItemListener` interface. The `HandleItemEvent` class is instantiated in line 39 and is bound to the three checkboxes in lines 40, 41 and 42. The example demonstrates how a single handler can be bound to multiple sources. To identify which checkbox has fired an item event we use the method `getItemSelectable()`, as shown in line 53. Once we know the source, we update the index accordingly as seen in lines 54 to 60. To know whether the item in the widget has been selected or deselected, we use the method `getStateChange()`. The return value of this method can be compared with the `SELECTED` and `DESELECTED` constants defined in `ItemEvent`. Both the methods `getItemSelectable()` and `getStateChange()` are also defined in the `ItemEvent` object that is passed as an argument to the event handler.

### *23.6.3 Keyboard Event Listener*

Key events are generated whenever the user presses and releases keys on the keyboard.

As described earlier, key events are low-level events and all widgets can register for key events. For widgets to receive key events it is essential that the widget has focus. To handle key events, we need to define a class that implements the `KeyListener` interface. The `KeyListener` interface defines three functions:

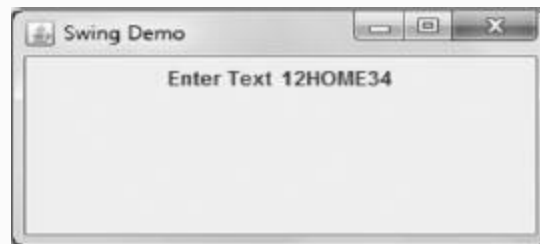
**`keyPressed(KeyEvent key)`:** This method is invoked when the user presses any key on the keyboard.

**`keyReleased(KeyEvent key)`:** This method is invoked when the user releases the pressed key on the keyboard.

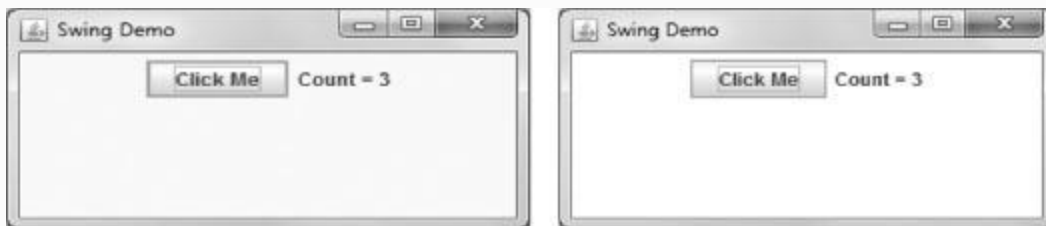
**`keyTyped(KeyEvent key)`:** This method is invoked whenever the user presses a printable key such as the characters a,b,c or symbols such as +, \_ , or \* on the keyboard. The difference between `KeyPressed` and `KeyTyped` methods is that `KeyTyped` method is invoked only when user types printable keys while

KeyPressed method is invoked for all the keys such as function keys F1, F2, HOME, INSERT and DELETE keys.

Example 23.13 illustrates how to create a key event handler and handle key events. The output is shown in [Figure 23.14](#).



**Figure 23.14** Keyboard event



**Figure 23.15** Output of [Example 23.14](#)



## Example 23.13: Program to Demonstrate Key Event Handling

```
1. import java.awt.*;
2. import java.awt.event.KeyEvent;
3. import java.awt.event.KeyListener;
4. import javax.swing.*;
5. public class SwingDemo {
6.     public static void main(String[]
args) {
7.         JFrame frame = new
DisplayFrame();
8.         frame.setVisible(true);
9.     }
10. }
11. class DisplayFrame extends
JFrame{
12.     DisplayFrame() {
13.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
14.         setTitle("Swing Demo");
15.         setSize(300, 150);
16.         setLocation(100,100);
17.         JPanel panel = new
DisplayPanel();
18.         getContentPane().add(panel);
```

```

19.  }
20.  }
21.  class DisplayPanel extends
JPanel{
22.      String msg = "";
23.      JLabel label;
24.      DisplayPanel() {
25.          label = new JLabel(msg);
26.          label.addKeyListener(new
HandleKeyEvent());
27.          label.setFocusable(true);
28.          JLabel disp_label = new
JLabel("Enter Text");
29.          add(disp_label);
30.          add(label);
31.      }
32.      class HandleKeyEvent implements
KeyListener{
33.          public void keyPressed(KeyEvent
ke) {
34.              int key = ke.getKeyCode();
35.              switch(key)
36.              {
37.                  case KeyEvent.VK_HOME:
38.                      msg += "HOME";
39.                      break;
40.                  case KeyEvent.VK_INSERT:
41.                      msg += "INSERT";
42.                      break;
43.                  case KeyEvent.VK_DELETE:
44.                      msg += "DELETE";
45.                      break;

```

```
46.    case KeyEvent.VK_END:
47.        msg += "END";
48.        break;
49.    }
Listener
50.        label.setText(msg);
51.    }
52.        public void
keyReleased(KeyEvent ke)
53.        {
54.        }
55.        public void keyTyped(KeyEvent
ke)
56.        { msg += ke.getKeyChar();
57.            label.setText(msg);
58.        }
59.    }
60. }
```

---

The application above updates a label dynamically each time the user enters a key on the keyboard. Updating the label is done in the class `HandleKeyEvent` defined in line 32 which implements the `KeyListener` interface. The `HandleKeyEvent` class is instantiated and bound to the label in line 26. As discussed earlier, it is mandatory for widgets wanting to receive key events to

have focus. This is achieved using the method `label.setFocusable(true)` in line 27. In case of a non-printable keys, we use the method `getKeyCode()` line 34 to retrieve the key pressed and in the case of printable keys we retrieve the key using the method `getKeyChar()` in line 56. Both the methods are defined in KeyEvent object which is passed as an argument to the event handler functions. As seen from the code, the method `getKeyCode` returns an integer constant. We use a switch statement to identify and process the special keys. The function `keyReleased()` is a dummy function but he have to define it as it as specified in the interface else we will get compilation errors.

#### *23.6.4 Mouse Event Listener*

Mouse events like the key events are low-level events. To handle the mouse events, we need to define a class that implements the `MouseListener` interface. The `MouseListener` interface defines five functions

**mouseEntered(MouseEvent e):** This method is invoked when the cursor enters the widget area.

**mouseExited(MouseEvent e):** This method is invoked when the cursor leaves the widget area.

**mousePressed(MouseEvent e):** The method is invoked when the user presses the mouse button.

**mouseReleased(MouseEvent e):** The method is invoked when the user releases the mouse button.

**mouseClicked(MouseEvent e):** The method is invoked when the user clicks, that is, presses and releases, the mouse button.

Example 23.14 illustrates how create a mouse event handler and handle mouse events. Figure 23.15 shows the output.

## Example 23.14: Program to Demonstrate Mouse Event Handling

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. public class SwingDemo {
5.     public static void main(String[]
args) {
6.         JFrame frame = new
DisplayFrame();
7.         frame.setVisible(true);
8.     }
9. }
10. class DisplayFrame extends
JFrame{
11.     DisplayFrame(){
12.
setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
13.         setTitle("Swing Demo");
14.         setSize(300, 150);
15.         setLocation(100,100);
16.         JPanel panel = new
DisplayPanel();
17.         getContentPane().add(panel);
18.     }
```

```
19. }
20. class DisplayPanel extends
JPanel{
21.     int count;
22.     JLabel label;
23.     JButton button;
24.
25.     DisplayPanel() {
26.         count = 0;
27.         button = new JButton("Click
Me");
28.         label = new JLabel();
29.         label.setText("Count = "
+count);
30.         button.addMouseListener(new
HandleMouseEvent());
31.         add(button);
32.         add(label);
33.         setBackground(Color.WHITE);
34.     }
35. class HandleMouseEvent implements
MouseListener{
36.     public void
mouseClicked(MouseEvent e) {
37.         count++;
38.         label.setText("Count = "
+count);
39.     }
40.     public void
mouseEntered(MouseEvent e) {
41.
42.         setBackground(Color.YELLOW);
```

```
43.  }
44.  public void
mouseExited(MouseEvent e) {
45.      setBackground(Color.WHITE);
46.
47.  }
48.  public void
mousePressed(MouseEvent e) {
49.
50.  }
51.  public void
mouseReleased(MouseEvent e) {
52.
53.  }
54.  }
55. }
```

---

In this example, we handle two tasks: one is count the number of times the user has clicked the button “Click Me” and the second involves setting the background colour of the panel Yellow each time the cursor is in the button area. To achieve these tasks, we use a mouse event handler called `HandleMouseEvent` which implements the `MouseListener` in line 35. The `HandleMouseEvent` is instantiated and bound to the button widget in line 30. As



seen from the code, we update the count and set the Label in lines 37 and 38. The colour of the panel is changed using the method *setBackground( )*, as seen in lines 42 and 45.

## 23.7 Applets

Applets are small java programs that are transported from one computer system on the network to another system. They cannot be run independently but they need web browser or program called appletviewer to run on the system.

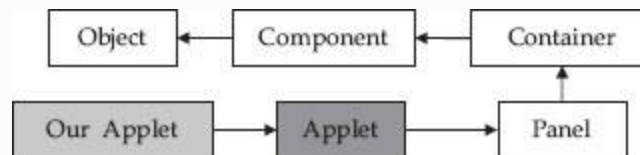
An applet, just like an application, can perform lot of tasks such as computation, taking user inputs, passing parameters, display using Graphics-rich features, play music and last but not least create animations. But the beauty is that applet can travel on computer from one location to another embedded onto HTML Tags when user imports them using Internet.

### *23.7.1 Concepts of Applets*

An applet is a java program that requires another host program to run it, i.e., Applet program gets embedded in another program and gets executed. Applets are run from web browsers such as IE or Google Chrome.

Applet programming is Java's way to provide the user with excellent graphics and GUI facilities, in addition to other conveniences such as distributed and object-oriented features.

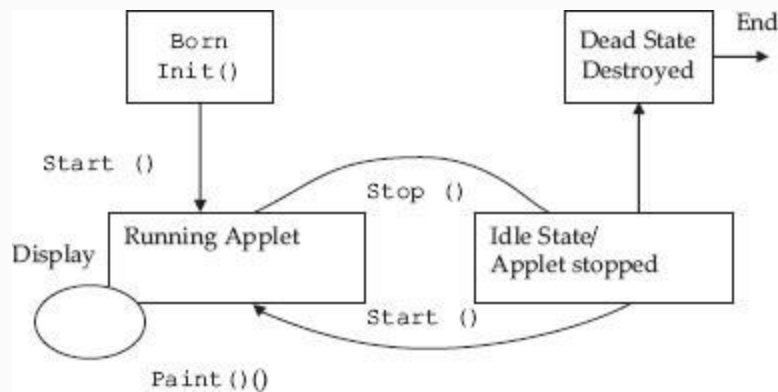
We have Applet class with hierarchy java->awt->Applet. So the applet class we are going to write must extend to package java.applet.Applet. The details are in Figure 23.16.



**Figure 23.16** Class hierarchy of applets in java

### *23.7.2 Life Cycle of an Applet*

There are four states in a life of an applet: Born, Running, Idle and Dead. They are explained in the diagram shown in Figure 23.17.



**Figure 23.17** Life cycle of an applet

**Initialization:** Applet when it is born calls `init()` method that sets initial conditions such as create objects needed, set font and colour and images, set up sizes of window and set up initial values. Usually we need to write our own `init()` method to override base class `init()` method.

**Running:** When an applet is created, it enters a Running state when it calls for a

`start()` method. `Start()` method is called automatically on initialization of applet by `init()` method. An applet which is in idle state can also be started. Therefore, we can say that `start()` method can be called any number of times.

**Idle/Stop().** An applet goes to idle state when a `stop()` method is called. This `stop()` is called automatically whenever web page running the applet is left. If an applet is running on a thread, then the thread will be stopped.

**Dead state:** It is a state when an applet is removed from the main memory. A method called `destroy()` is used to clean up any memory spaces.

**Display state:** This occurs when applet runs `paint()` method or a panel object is added to Container object as in the case of Applets with swing components.

To summarize, the methods that are invoked when an applet or JApplet is loaded

by any browser, there are four methods which are automatically initiated by the browser. They are

- `Init()`: Initiated automatically whenever an Applet is loaded. It is like a constructor. It initially creates the applet including initialization of threads.
- `Start()`: Automatic whenever `init()` is called.
- `Stop()`: Invoked automatically whenever `destroy()` method is called.
- `Destroy()`: Invoked automatically whenever browser is terminated. Threads have to be stopped and removed in this section.
- `Paint(Graphics g)`: Invoked automatically whenever `start()` is called.
- `Update(Graphics g)`: After `paint()`, `update()` updates the container.

`Repaint(Graphics g)`: invokes `update()` which calls `paint()` which in turn calls `update()` to update the container.

### *23.7.3 Creating an Applet – HelloWorld Applet*

In our first example, we will write an applet program to highlight the issues and commands we use to set applet work. Applet Programming has two distinct stages:

- Develop Applet Program and compile just like ordinary java application. Output will be HelloWorldApplet.class.
- Write HTML (Hyper Text Mark Up Language) Tags shown below in any text editor and name the file as HelloWorldApplet.html. Embed applet tag in the html file.
- Execute the applet program by using appletviewer or web browser. Details are shown below.

## Example

**23.15: HelloWorldApplet.java  
With Html Tags Embedded in the  
Main Java Programme. This Code  
Can Only be Executed with  
Appletviewer**

---

```
1.  /* <HTML>
2.  <BODY>
3.  <Applet
Code="HelloWorldApplet.class" Height=320
Width=400>
4.  </Applet>
5.  </BODY>
6.  </HTML>
7.  */
8.  import java.awt.*;
```

```
9. import java.applet.*;
10. import java.awt.Graphics.*;
11. public class HelloWorldApplet
extends Applet {
12. public void paint(Graphics g) {
13. g.drawString(«Hello There !This
is how Applets look & feel !», 65 , 95);
14. }
15. }
```

---

**Compilation:** Go to the Directory where the java file has been entered.

```
javac HelloWorldApplet.java
```

```
Appletviewer
```

HelloWorldApplet.java Output is shown in Figure 23.18



**Figure 23.18** An applet output

## Example

**23.16: HelloWorldApplet.java with HTML File Separately Created and Class File is Embedded in HTML File. This Code Can be Executed with AppletViewer or Web Browser.**

```
//HelloWorldApplet.java
1. import java.awt.*;
2. import java.applet.*;
3. import java.awt.Graphics.*;
4. public class HelloWorldApplet
extends Applet {
    5. public void paint(Graphics g) {
    6. g.drawString("Hello There !This is
how Applets look & feel !", 65, 95);
    7. }
    8. }
```

**Compile: javac  
HelloWorldApplet.java**



In the same directory create a separate html file called HelloWorldApplet.html

---

```
1. <HTML>
2. <BODY>
3. <Applet
Code="HelloWorldApplet.class" Height=320
Width=400>
4. </Applet>
5. </BODY>
6. </HTML>
```

---

## Running of program:

Appletviewer  
HelloWorldApplet.html

Or go to icon of web browser in the directory where html file has been created and double press on the icon.

The **import** statements direct the Java compiler to include the `java.applet.*` and `java.Graphics.*` As Applet classes use extensively the features of windows, we also need to import a package called awt that stands for Advanced Windowing ToolKit (awt).

The Hello class extends Applet class; the Applet class provides the framework for the host application to display and control the lifecycle of the applet. The Applet class provides the applet with the capability to display GUI and respond to user events.

The Hello class overrides the `paintComponent(Graphics)` method provide the code to display the applet. The `paint()` method is passed a Graphics object that contains the graphic context used to display the applet. The `paintComponent()` method calls the graphic context `drawString(String, int, int)` method to display the “Hello there! This is how Applets look and feel” string at a pixel offset of (65, 95) from the upper-left corner in the applet's display.

**HTML Document:** In order to run HelloWorldApplet, you need another program called `HelloWorldJApplet.html`, which is essentially a program that contains `HelloWorldJApplet` class. HTML stands

for hypertext mark up language. HTML is the language, called scripting language, that the browser understands.

## **A typical HTML Document looks like this. Call this: HelloWorldApplet.html**

---

```
<html>
<head>
<title>Hello World Applet</title>
</head>
<body>
<applet code="HelloWorldApplet"
width="200" height="200">
</applet>
</body>
</html>
```

---

An applet is placed in an HTML document using the **<applet>** HTML element. The applet tag has three attributes set: **code="HelloWorldApplet"** specifies the name of the Applet class and **width="200"** **height="200"** sets the pixel width and height of the applet.

The host application, typically a Web browser, instantiates the

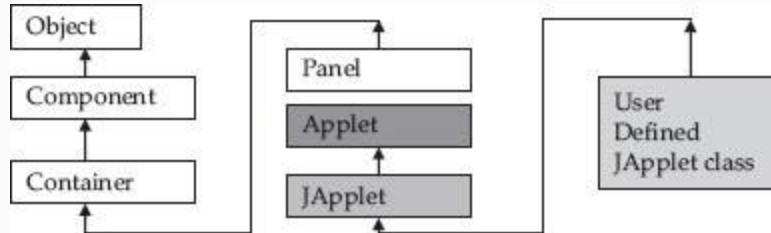
**HelloWorldApplet** applet and creates an AppletContext for the applet. Once the applet has initialized itself, it is added to the AWT display hierarchy. The `paint()` method is called by the AWT whenever the display needs the applet to draw itself.

#### *23.7.4 JApplets – Applets with Swing Components: Better Look and Feel*

Java has introduced the latest Java Swing Components with class hierarchy: `java->awt->Applet->JApplet`. In order to get better and “feel good” features, it is better we make our Applet class to extend to `JApplet` instead of `Applet`. As most of the Applets use Window features, we need to import advanced windowing tool kit provided by Java and within package: `java.awt`. Swing components are provided by `javax` (x stands for java extended) to provide rich libraries so that GUI/Interfaces, etc., are easily implemented.

We can clearly see in Figure 23.19 that `JApplet` extends `Panel` that in turn extends

to Container class. Therefore, we need to add components to JApplet class.



**Figure 23.19** Class hierarchy of JApplet class

## Example

### 23.17: HelloWorldJApplet.java: Java Applet with Swing

```
import javax.swing.*;
import java.awt.*;
public class HelloWorldJApplet extends
JApplet {
public void init() {
//First Get Container object
Container
myContentPane=getContentPane();
```

```
// we have set the layout. we will use
flow layout
myContentPane.setLayout(new
FlowLayout());
//Prepare the lable to post on
myContentPane
JLabel message = new JLabel("Welcome to
swing components in Java");
//add label to myContentpane
myContentPane.add(message);
}
}
```

<b>Line No. 7:</b>	defines an object of Container class called myContentpane. getContentPane() returns object of Container.
<b>Line No. 9:</b>	is setting the layout to FlowLayout()
<b>Line No. 11:</b>	creates an object of JLabel and passes a Tring for display.

<b>Line No 13:</b>	Just adds message to Container.
--------------------	---------------------------------

To obtain the output, compile the program to get `HelloWorld.class`. If you are using eclipse IDE you can directly execute the program to get the output. Or create an HTML document as explained in the previous section. The output is shown in Figure 23.20.



**Figure 23.20** JApplet output

## Example

### 23.18: HelloWorldJApplet.html: Html Script for HelloWorldJApplet.java

```
<html>
<title>HelloWorldJApplet</title>
<body>
  <applet code="HelloWorldJApplet.class"
    width=400 height = 200>
  </applet>
</body>
</html>
```

Create the above program by name  
HelloWorldJApplet.html in the  
directory  
c:\oopsjava\workspace\oopstech2\  
src\com\oops\chap23 or any other  
directory you have been working.  
appletviewer.exe is the program supplied by  
java to test applets.



```
C:\Oopsjava\examples\bin\>  
appletviewer
```

HelloWorldJApplet.java will produce the output shown above. Alternatively, you can also create separate html file and use web browser like IE to get the same output by simply double clicking on HelloWorldJApplet.html file to invoke the web browser.

The **import** statements direct the Java compiler to include the `javax.swing.JApplet` and `java.Graphics.*`. As JApplet classes use extensively the features of windows, we also need to import a package called `awt`.

The Hello class extends JApplet class; the JApplet class provides the framework for the host application to display and control the lifecycle of the applet. The JApplet class provides the applet with the capability to display GUI and respond to user events.

Which is better Applet or JApplet in terms of look and feel? Refer to Figures 23.18 and 23.20. You will agree that applets with swing components offer much better look and feel. Hence we will use swing components with

Applets. From now on we will not be writing HTML Tags. It is understood that you will be able to understand and execute your applet programs.

### *23.7.5 Applets vs. Stand-alone Applications*

#### **Stand-Alone Applications**

The programs we did so far are called stand-alone programs. The signature of stand-alone applications is that it will have a `main()` method. Stand-alone application in java is similar to application development using c++ and contains all features of language.

#### **Applets**

- Applets are run by web browsers. They are designed to work on the Internet.
- The signature of applet is that it will not have `main()` methods. Instead it will have `init()` methods.
- Applets cannot be run independently. They need to be embedded into HTML tags. HTML stands for Hyper Text Mark Up Language.
- Due to security restrictions, applets cannot read from local files/disks or write to a disk.
- Applets do not have any access to any other programs residing in local computers.

### 23.7.6 *Passing Arguments to Applets*

The applet tag we have used to embed out class in HTML tags is shown below:

```
<applet  
code="HelloWorldApplet"  
width="200" height="200">  
</applet>.
```

This is a simple tag. We will now introduce you to a full-fledged HTML file.

#### **Example 23.19: An Example HTML Full Fledged File**

```
<APPLET  
[CODEBASE = codebase_URL] //  
directory of html file  
CODE= "Applet class name" // for  
example "HelloWorldJApplet.class"  
[ ALT = alternate_text]  
[ NAME = applet _instance_name]  
WIDTH=no of pixels  
HEIGHT = no of pixels  
[ ALLIGN = alignment] //TOP BOTTOM,
```

```
LEFT,RIGHT,MIDDLE etc
    [VSPACE]= pixels] // used when align
LEFT or RIGHT is used
    [HSPACE]=pixels] //used when TOP or
BOTTOM are used
>
[PARAM NAME = name1 VALUE = value1> ]
// used for parameters passing
[PARAM NAME = name2 VALUE = value2> ]
</APPLET>
```

---

Let us attempt an example wherein we pass arguments to applet. We will pass colour and text to be displayed by the applet. In addition, we will pass two numbers and operations to be performed by the applet. Let us see the result.

**Example 23.20: Write a Swing-based Applet to Accept String Value as Parameter From Html Tag File. The Program to Display Hello India! If There is Input India From the Tag File. Else it Has to Display Hello Delhi!**

---

```
//: HTMLParam.java
1. /*<HTML><BODY><Applet
Code="HTMLParam.class"
2. width=400 height =200> <PARAM NAME
= "String"VALUE= "India!" >
3. </Applet></BODY></HTML> */
4. import javax.swing.*;
5. import java.awt.*;
6. public class HTMLParam extends
JApplet {
7. String stg;
8. public void init() {
9. //Get the String to be displayed
10. stg=getParameter("String");
11. if ( stg==null)
12. stg="Delhi!";
13. stg="Hello "+stg;
14. //First Get Container object
15. Container
myContentPane=getContentPane();
16. // we have set the layout. we
will use flow layout
17. myContentPane.setLayout(new
FlowLayout());
18. //Prepare the lable to post on
myContentPane
19. JLabel message = new JLabel(stg);
20. //add label to myContentpane
21. myContentPane.add(message);
```

```
22. }  
23. }
```

---



## Example

**23.22: InputToApplet.java A Program to Accept Bank Account and Transaction Amount and Applet Will Return Bank Balance After Transaction**

---

```
/*<HTML><BODY><Applet  
Code="InputToApplet.class" width=800  
height =800></Applet></BODY></HTML> */  
1. package com.oops.chap23;  
2. import javax.swing.*;  
3. import java.awt.*;
```

```

4. import java.awt.event.*; // for
action listener interface
5. public class SwingInputs extends
JApplet{
6. public void init() {
7. //First Get Container object
8. Container
myContentPane=getContentPane();
9. // we have set the layout. we will
use flow layout
10. myContentPane.setLayout(new
GridLayout(1,1));
11. //Prepare the lable to post on
myContentPane
12. JPanel panel = new
SwingPanel();
13. myContentPane.add(panel);
14. }
15. }
16. class SwingPanel extends
JPanel implements ActionListener{
17. String val1,val2,stg;
18. double bal=20000.00;
19. JTextField jtf1,jtf2,jtf3;
20. SwingPanel(){
21. JLabel label =new
JLabel("Enter accountNo :");
22. add(label);
23. jtf1=new JTextField(10);
24. jtf1.addActionListener(this);
25. add(jtf1); // converts jtf1 to
component

```

```

26.      setLocation(100,100);
27.      label =new JLabel("Enter
Transaction Amount : ");add(label);
28.      jtf2=new JTextField(10);
29.      jtf2.addActionListener(this);
30.      add(jtf2);
31.      label =new JLabel("\nBank
Balance ");add(label);
32.      jtf3=new JTextField(10);
33.      stg=String.valueOf(bal);
34.      jtf3.setText(stg);
35.      add(jtf3);
36.      label = new JLabel("Have a
Great Day ahead!");
37.      add(label);
38.      }
39.      public void
actionPerformed(ActionEvent e){
40.          int
acn=Integer.parseInt(jtf1.getText());
41.          double
amt=Double.parseDouble(jtf2.getText());
42.          double banbal=bal-amt;
43.          String
stg=String.valueOf(banbal);
44.          jtf3.setText(stg);
45.      }
46.      }

```

---



<b>Line No. 4:</b>	imports <code>java.awt.event.*</code> ; // for action listener interface
<b>Line No. 8:</b>	defines myContentpane as object of Container.
<b>Line No. 12:</b>	defines a JPanel object called panel. We can add components now to this panel.
<b>Line No. 16:</b>	class <u>SwingPanel</u> extends JPanel implements ActionListener{ Implements action listeners for attaching action listeners events to widgets inside the class.
<b>Li</b>	declares three JText fields ; <code>JTextField</code>

<b>Line No. 19:</b>	<code>jtf1,jtf2,jtf3;</code>
<b>Line No. 20:</b>	<p>is a constructor SwingPanel. Line No. 21 to 38 are used to define the text fields and attach action Listeners by using the method <code>addActionListeners</code> like <code>jtf1.addActionListener(this);</code> at Line No. 23.</p>
<b>Line No. 39:</b>	<p>defines a mandatory method called <code>public void actionPerformed(ActionEvent e) {</code></p>
<b>Lines 40 &amp;</b>	<p><code>int acn=Integer.parseInt(jtf1.getText()); double amt=Double.parseDouble(jtf2.getText());</code> are used to get the text in String format and convert them to basic data type using <code>parseInt</code> and <code>parseDouble</code>.</p>

41  
:

Line  
No.  
42:

displays the amount by converting basic data type to String type . It uses `valueOf()` method



## 23.8 Summary

1. Graphical User Interface (GUI) development in JAVA is based on components. Java components can further be divided into containers and widgets.
2. Containers are further classified as Root Containers and Intermediate Containers.
3. Containers also called as Top level containers are necessary for every GUI program. Examples are three top-level containers: JFrame, JDialog and JApplet.

4. Intermediate containers are not mandatory but are used widely as they help in managing and organizing widgets. JPanel is the most widely used intermediate container in Swing.
5. Widgets are GUI elements which are used to interact with the user. Widgets are also used to get user inputs. JButton, JLabel, JScrollBar and JCheckBox are examples of widgets in Swing.
6. For Swing Applications, the root is always JFrame.
7. Panels are intermediate containers and as such not essential for creating Java graphical user interface programs.
8. Widgets are the components which help us to build user interfaces. They are the components that the user actually sees and interacts with users.
9. The JLabel object is used to display text, images or both text and images in Java.
10. The JButton object is used to display buttons in Java. Buttons are very useful user interface elements, which can be used for a variety of user interactions.
11. The JCheckBox object is used to display checkboxes in Java. Checkboxes allow users to either select or deselect an item; they also allow multiple items to be selected at the same time.
12. The JComboBox in Java has a button and a drop-down list.
13. Menus in Java are implemented using three components, namely, JMenuBar, JMenu and JMenuItem.
14. Border layout divides the panel into five regions North, South, East, West and Center, as shown below.
15. Grid Layout divides the screen into specified rows and columns. Grid layout is useful when we want to create UI similar to windows desktop wherein icon is symmetrically placed in the screen.

16. Flow layout is the default layout. If you do not specify any layout using the *setLayout()* method, then by default the panel will have Flow layout. In flow layout, the widgets are arranged from left to right and then from top to bottom.
17. Events are generated due to user activity and they can range from key presses, mouse clicks, to selecting/deselecting checkboxes and button clicks.
18. Toolkit (AWT) framework is responsible for delivering user-generated events to the applications. AWT also defines the interfaces for all the event handlers.
19. Events in Java can be classified into two types: high-level events and low-level events.
20. High-level events such as Action events are specific to certain widgets and are sent to the application only if those widgets are present and have registered for the event.
21. Low-level events such as Keyboard and Mouse events are sent to all the widgets which have registered to receive them.
22. Action events are generated whenever the user performs any action on the widget like pressing a button or choosing an item from the menu.
23. To enable us to handle action events, we need to create a class that implements the ActionListener interface. The ActionListener interface requires us to implement just one method *actionPerformed(ActionEvent e)*.
24. Item events are generally fired by components such as checkboxes, radio buttons and combo boxes wherein the user selects or deselects an item.
25. ItemListener interface requires us to implement one method *itemStateChanged(ItemEvent e)*.
26. Key events are generated whenever the user presses and releases keys on the Keyboard. The events are:  
`keyPressed(KeyEvent key) :`

```
keyReleased(KeyEvent key) :
```

```
keyTyped(KeyEvent key) .
```

27. Mouse events like the key events are low-level events.

They are `mouseEntered (MouseEvent e) :`

```
mouseExited(MouseEvent e) :
```

```
mousePressed(MouseEvent e) :
```

```
mouseReleased(MouseEvent e) :
```

```
mouseClicked(MouseEvent e) .
```

28. Applets are small java programs that are transported from one computer system on the network to another system. They cannot be run independently but they need web browser or program called appletviewer to run on the system.

29. Applet class hierarchy is: `java->applet->Applet`.

So the applet class must extend to package

```
java.applet.Applet.
```

30. When applets are loaded web browser automatically calls the methods `init()`, `Start()` : `Stop()` :

```
Destroy() : Paint(Graphics g) :
```

```
Update(Graphics g) .
```

31. Java has introduced the latest Java Swing Components with class hierarchy: `java->applet->Applet->JApplet`. In order to get better and “feel good” features.

32. Applets cannot be run independently. They need to be embedded into HTML tags. HTML stands for Hyper Text Mark Up Language.

33. Due to security restrictions, applets cannot read from local files/disks or write to a disk. Applets do not have any access to any other programs residing on local computers.

34. When local disks need to be handled, applications supported by JFrame provide excellent graphics facilities.

# Exercise Questions

## Objective Questions

1. Which component is not part of root containers?

1. JDialog
2. JApplet
3. JFrame
4. JPanel

2. Which of the following are not part of widgets?

1. JPanel
2. JButton
3. JTextField
4. JMenu

3. JFrame is a

1. Top-level container
2. Intermediate container
3. A widget
4. Component

4. Which of the following are not mandatory?

1. top-level container
2. intermediate
3. widget
4. components

5. Which of the following are offered by JFrame?

1. title
2. icon
3. set height width and location
4. add widgets directly

1. i and ii
2. ii, iii and iv
3. i, ii, iii and iv
4. i, ii and iii

6. JPanel is a container belonging to

1. top level
2. bottom level
3. widget
4. intermediate level

7. The component used for displaying the text in a window

1. JPanel
2. JLabel
3. JTextField
4. JFrame

8. The layout to be used when we need rows and columns on screen is a

1. Flow layout
2. Grid layout
3. Border layout
4. Matrix layout

9. Action Listeners needs actionPerformed() method to be executed.

TRUE/FALSE

10. ItemListener requires the following method to be implemented:

1. itemStateChanged(ItemEvent e)
2. addActionListener(new HandleEvent())
3. actionPerformed(ActionEvent e)
4. getItemSelectable()

### Short-answer Questions

11. What are the two packages required to do multimedia programming in java?
12. What are high-level containers?
13. What are widgets?
14. What are intermediate containers?
15. Distinguish Applet and JApplets.
16. Distinguish Action Listeners and Item Listeners?
17. Explain the states of n Applet.
18. Explain Applet Tag and mandatory tags.



19. Explain how parameters are passed to Applet through HTML files.
20. Distinguish Applets and Stand-alone applications.

#### **Long-answer Questions**

21. Distinguish between Component and Container.
22. Explain the class hierarchy of Applet and JApplet.
23. Explain the methods in Keyboard event listeners.
24. Explain the methods in Mouse Event listeners.
25. Explain how an applet program can be converted into application Program.

#### **Assignment Questions**

26. Develop an application program with swing components for recording of students results on a remote computer (Server). The data obtained through a form needs to be submitted online for the server to record it on to its random file.
27. Modify the above program so that a student can submit his roll number and he is informed about his result. Use Applet with swing components for developing interface.
28. Develop an interface wherein a customer can select his choice of items in a restaurant using JCheckBoxes and a Button and he is informed of the cost of his selection.
29. Write a program to implement calculator with basic arithmetic operation.
30. Write a multithreaded Applet for analog clock second, minutes and hour handles.

### **Solutions to Objective Questions**

1. d
2. a
3. a
4. b

- 5. c
- 6. d
- 7. c
- 8. b
- 9. True
- 10. a

16. b

## Collections and Software Development Using Java

### LEARNING OBJECTIVES

*At the end of this chapter, you will be able to understand*

- Collections offered by Java through collection class interfaces such as `Set<T>`, `List<T>`, `Queue<T>`, and `Map<k,s>`.
- Understand Java database connectivity and tools and drivers offered by Java and other vendors such as Microsoft, MySQL.com, and Oracle.
- Develop applications using Oracle, MySQL, and access databases.
- Understand and deploy the concepts underlying Servlets in Client Server programming of Java.
- Understand and deploy the concepts underlying Java Beans in Client Server programming of Java.

## 24.1 Introduction

While well-designed algorithms improve the efficiency of programs, the data structures will allow a programmer to use the underlying hardware and supporting language features to enhance the throughput of the programmers. J2SE 5.0 and later versions have provided collection framework that provides several interfaces such as `Set<T>`, `List<T>`, `Queue<T>`, and `Map<k, s>`. This chapter introduces you to these concepts so that you can deliver programs by efficiently using collections rather than developing programs yourself, thereby enhancing productivity.

In industry, databases are used extensively because of ease with which they provide answers to queries without resorting to elaborate programming through a software called sql query. The most popular databases in the industry are Oracle, MySql (open source by [MySql.com](http://MySql.com)), and, of course, MS Access that is within easy reach through

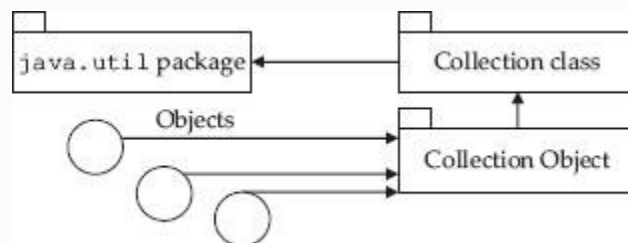
all prevalent MS Office. This chapter shows you ways to handle these databases through Java programs by using drivers such as JDBC, JDBC-ODBC, and MySql connectivity tools. Servlets are important implementation in Java to handle client server programs. Servlets extend the functionality of web servers to any user on the net just like applets extend the functionality of the web browser. Java Beans is designed for reusability. In this chapter, you will learn the procedures to deploy java beans.

This chapter will put you through various skills that are required to become highly productive and make you ready to take up a job in the industry or enhance your skills. We teach you the methodologies through step-by-step procedures and programs and reflect these steps thereby making learning permanent and easy.

## 24.2 Collection Framework of Java

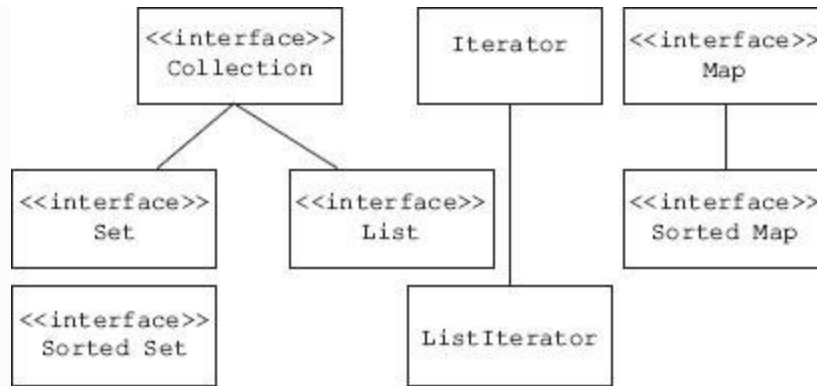
How do we store a group of objects in an array? J2SE 5.0 has provided a collection

framework. Collection framework provides an API that makes programming using data structures easy and thereby enhances the productivity of the programmer. This is somewhat similar to the container of C++ but has quite a number of differences. Collection framework is implemented in java.util package. The concept is make a collection object, store group of objects in collection object. The collection framework of Java is shown in Figure 24.1



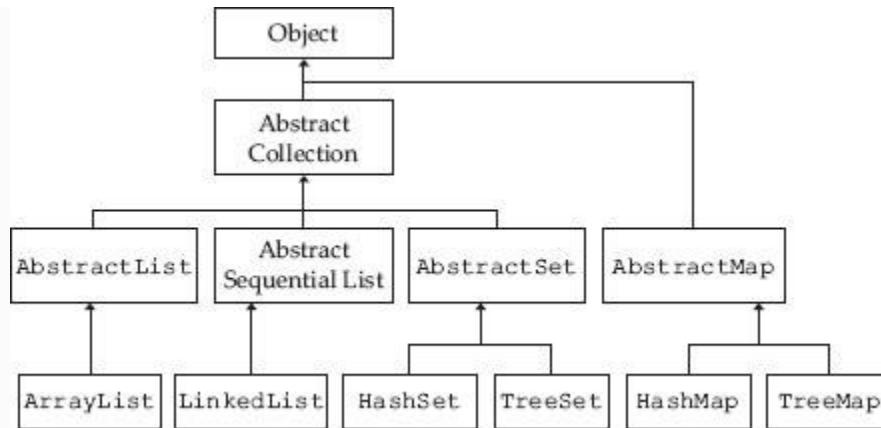
**Figure 24.1** Collection class of java util

Collection class provides several interfaces such as Collection, Map, and Iterator. These are shown in Figure 24.2.



**Figure 24.2** Interfaces provided by java collection

The Collection Framework classes and interfaces have been derived from meta class Object provided by Java. The hierarchy of classes of Collection framework is shown in Figure 24.3.



**Figure 24.3** Class hierarchy of collection framework classes

Interfaces and their implementing class are shown in Table 24.1.

**Table 24.1** Collection class interfaces and implementing classes



Interface	Implementing Class	Description
List	ArrayList	Implementation based on resizable array
List	LinkedList	Implementation based on linked list
Set	HashSet	Implementation based on Hash table
Set	TreeSet	Implementation based on balanced binary tree
Map	HashMap	Implementation based on Hash table
Map	TreeMap	Implementation based on balanced binary tree

## 24.3 Collection Interface

The Collection Interface is the main interface from which the List and Set Interfaces are derived. It specifies the methods that are common to all Collections. The methods defined in the Collection Interface and their descriptions are listed in [Table 24.2](#).

**Table 24.2** Collection Interface and their descriptions

Method	Remark
--------	--------

boolean add(Object o)	Add a new object
boolean addAll(Collection c)	Usage: t.addAll(c) Mimics union operation.  Result: Collection t contains all the objects in both t and c
void clear()	Remove All elements in the collection
boolean contains(Object o)	Checks if object is present in collection
boolean containsAll(Collection c)	Usage: t.containsAll(c) Mimics subset operation  Result: returns true if Collection c is subset collection t
boolean isEmpty()	Returns true if collection is empty
Iterator iterator()	Used to traverse the collection
boolean remove(Object o)	Remove object from collection
boolean	Usage: t.removeAll(c)

<code>removeAll(Collection c)</code>	Result: All objects in Collection <code>t</code> which are also contained in collection <code>c</code> are removed
<code>boolean retainAll(Collection c)</code>	Usage: <code>t.retainAll(c)</code> Mimics intersection operation.  Result: Collection <code>t</code> has objects common to both <code>t</code> and <code>c</code>
<code>int size()</code>	Number of objects in the collection
<code>Object[] toArray()</code>  <code>Object[] toArray(Object[] a)</code>	The array operations allow the contents of a Collection to be translated into an array

## 24.4 Set Interface

Important features of Set interface are as follows:

- Defined in *java.util.Set*
- Set extends the Collection Interface but does not add any new methods
- The interface models the mathematical set abstraction
- Set cannot contain duplicate elements
- Collection framework provides two implementations of Set Interface: `HashSet` and `TreeSet`

We demonstrate some basic set operations in Example 24.1 using the HashSet implementation of the Set interface.

### **Example 24.1: Program to Demonstrate Basic Set Operations**

```
1. import java.util.Set;
2. import java.util.HashSet;
3. public class SetDemo {
4.     public static void main(String[]
args) {
5.         Set<String> PoolA=new HashSet<String>
();
6.         PoolA.add("India");
7.         PoolA.add("Australia");
8.         PoolA.add("South Africa");
9.         System.out.println("PoolA: " +PoolA);
10.        Set<String> PoolB=new
HashSet<String>();
11.        PoolB.add("England");
12.        PoolB.add("India");
13.        PoolB.add("New Zealand");
14.        System.out.println("PoolB: "
```

```

+PoolB);
15.
16. PoolA.retainAll(PoolB);
17. System.out.println("Intersection
Operation PoolA: " +PoolA);
18. PoolA.addAll(PoolB);
19. System.out.println("Union Operation
PoolA: " +PoolA);
20. System.out.println("Subset
Operation: "
21. +PoolA.containsAll(PoolB));
22. }
23. }
24. /* Output
25. PoolA: [Australia, South Africa,
India]
26. PoolB: [England, New Zealand, India]
27. Intersection Opeartion PoolA:
[India]
28. Union Operation PoolA: [England, New
Zealand, India]
29. Subset Operation: true
30. */

```

**Lines  
Nos.  
1 & 2:**

import Set and HashSet defined in the  
java.util package. Line 5 defines a set  
of strings called PoolA.

<b>Lines Nos. 6, 7 &amp; 8:</b>	show how to populate the Set <code>PoolA</code> using the add method.
<b>Line No. 9:</b>	demonstrates how we can print the elements to <code>PoolA</code> on the console.
<b>Lines Nos. 11–15:</b>	create a new Set <code>PoolB</code> and populate them with elements.
<b>Line No. 17:</b>	demonstrates the intersection operation between sets, Line 19 union operations and Lines 21–22 the subset operations.

## 24.5 Iterators

Iterator allows us to iterate over a collection such as `Set`. The iterator interface is defined in `java.util.Iterator`. The interface defines three methods as seen below:

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
}
```

```
public void remove();  
}
```

---

The `hasNext ( )` method returns true is the iteration has more elements. The `next ( )` method returns the next element in the collection and the `remove ( )` method is used to remove the last element returned by the iterator. Example 24.2 demonstrates usage of an iterator on a Set

### **Example 24.2: Program to Demonstrate Usage of Iterator**

---

```
import java.util.Set;  
import java.util.HashSet;  
import java.util.Iterator;  
public class SetDemo {  
    public static void main(String[] args)  
    {  
        Set<String> set=new HashSet<String>  
( );  
        set.add("Gautam");  
        set.add("Ramesh");  
    }  
}
```

```

        set.add("Anand");
        System.out.println("HashSet Output");
        for(Iterator it = set.iterator();
it.hasNext(); ){
            System.out.println("String: " +
it.next());
        }
    }
}
/*Output
HashSet Output String: Gautam String:
Ramesh String: Anand */

```

---

<b>L</b> <b>i</b> <b>n</b> <b>e</b> <b>N</b> <b>O</b> <b>.</b> <b>1</b> <b>2</b> <b>:</b>	shows how to obtain an iterator for the Set using the <code>iterator( )</code> methods. We use the <code>hasNext( )</code> method as the breaking condition for the loop as shown in Line 12. To get the value of the element we use the <code>next( )</code> method as shown in Line 13.
----------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



## 24.6 List Interface

Important features of List interface are as follows:

- Defined in *java.util.List*
- List extends the Collection Interface and adds a few new methods
- The elements are stored in a linear fashion with a beginning and an end.
- List can contain duplicate elements
- Collection framework provides two implementations of List Interface: `ArrayList` and `LinkedList`
- Lists have the capability to expand to accommodate the addition of new elements

Table 24.3 describes the additional methods defined by the List interface

**Table 24.3** Additional methods defined by List Interface

Method	Remark
--------	--------

E get(int index)	Returns element at given index
E set(int index, E element)	Replace value at given index with new element
boolean add(E element)	Adds element at the end of the list
void add(int index, E element)	Adds element at the given index
E remove(int index)	Removes element at the given index
int indexOf(Object o)	Returns index of object o in the list
int lastIndexOf(Object o)	Returns index of last found occurrence of Object o
ListIterator<E>	List Iterator extends the iterator it has ability to traverse the list in both directions, modify

<code>listIterator()</code>	the list and provide positional information such as current index
<code>List&lt;E&gt; subList(int from, int to)</code>	Returns a view of the list between indices from (inclusive) and to (excluded). Changes done in the sublist are reflected in the main list

We demonstrate some of the important methods of the List Interface and also usage of the List Iterator in Example 24.3:

### **Example 24.3: Program to Demonstrate Usage of List Interface and List Iterator**

```

1. import java.util.ArrayList;
2. import java.util.LinkedList;
3. import java.util.ListIterator;
4. public class ListDemo {
5.     public static void main(String[]
args) {
6.         ArrayList<String> AList = new
ArrayList<String>();
7.         LinkedList<String> LList = new
LinkedList<String>();

```

```
8. AList.add(new String("Ramesh"));
9. AList.add(new String("Anand"));
10. AList.add(new String("Gautam"));
11. System.out.println("ArrayList: "
+ AList);
12. DemoArrayList(AList);
13. LList.add(new String("Ramesh"));
14. LList.addFirst(new String("Anand"));
15. LList.addLast(new String("Gautam"));
16. System.out.println("LinkedList: "
+ LList);
17. System.out.print("Reversed
LinkedList: ");
18. ListIterator<String> it =
LList.listIterator(LList.size());
19. while(it.hasPrevious())
20. System.out.println(it.previous());
21. }
22. private static void
DemoArrayList(ArrayList<String> list){
23. String s = new String("Gautam");
24. System.out.println("index: "
+ list.indexOf(s));
25. list.remove(s);
26. System.out.println("ArrayList: "
+ list);
27. list.set(0, "Thunder");
28. System.out.println("ArrayList: "
+ list);
29. }
30. }
31. /* Output
```

```
32. ArrayList: [Ramesh, Anand, Gautam]
33. index: 2
34. ArrayList: [Ramesh, Anand]
35. ArrayList: [Thunder, Anand]
36. LinkedList: [Anand, Ramesh, Gautam]
37. Reversed LinkedList: Gautam Ramesh
    Anand
38. */
```

---

We instantiate two lists one is of the type `ArrayList (AList)` and the other is of the type `LinkedList (LList)` in Lines 6 and 7. Both the Lists hold Strings. Lines 9, 10, and 11 demonstrate how to add elements to an `ArrayList`. The function `DemoArrayList ( )` further demonstrates some of the important methods of the `List Interface`. Lines 15, 16, and 17 show the methods that can be used to add elements to a `LinkedList` as seen in addition to the `add` method `LinkedList` provides the `addFirst ( )` and `addLast ( )` methods which enable us to add elements to the starting and ending of the list easily. Line 21 shows how to create a `List iterator` and set to the last element in the list by passing the

`LinkedList.size()` parameter during instantiation. Lines 22 and 23 show how to iterate backwards.

## 24.7 Collection Algorithms

Java platform provides a number of useful algorithms. The algorithms are implemented as static methods in the `Collections` class. Many of the algorithms can be used only with Lists while some can be used with all the Collections. The algorithms are polymorphic in nature, i.e., the same method can be used on different implementations of a Collection interface. The algorithms can be used by importing the `java.util.Collections` class. In Table 24.4, we list some of the most widely used algorithms present in the collection class.

**Table 24.4** Collection algorithms description

Algorithm	Remark
-----------	--------

sort	Sorts the input list using an optimized merge sort algorithm. Worst case time complexity is $n\log(n)$ .
binarySearch	The binarySearch algorithm searches for a specified element in a List. The input list must be sorted before using the binarysearch algorithm
shuffle	Shuffles the order of elements in the list
reverse	Reverses the order of elements in a List
fill	Fills the List with the specified value.
copy	Copies elements in a source list to a destination list. The size of the destination list must be equal or greater than the source list.
swap	Swaps the elements at the specified positions in a List
addAll	Adds all the specified elements to a Collection. Generally elements are appended to the end of the Collection
frequency	Returns the number of instances of a specified element in a collection
disjoint	Used to determine if two collections have no elements in common

We demonstrate the usage of some the algorithms in Example 24.4.

### **Example 24.4: Program to Demonstrate Usage of Collection Algorithms**

```
1. import java.util.Collections;
2. import java.util.ArrayList;
3. public class ListAlgorithmDemo {
4.     public static void main(String[]
args) {

5.         ArrayList<Integer> IList = new
ArrayList<Integer>();
6.         ArrayList<String> SList = new
ArrayList<String>();
7.         IList.add(new Integer(7));
8.         IList.add(new Integer(5));
9.         IList.add(new Integer(9));
10.        Collections.sort(IList);
11.        System.out.println("Ascending Order:
" +IList);
12.        System.out.println("Index of Integer
```



```
9 in sorted list: "  
13. +Collections.binarySearch(IList,  
9));  
14. Collections.reverse(IList);  
15. System.out.println("Descending  
Order: " +IList);  
16. Collections.shuffle(IList);  
17. System.out.println("Shuffled List: "  
+IList);  
18. System.out.println("Min: "  
+Collections.min(IList));  
19. System.out.println("Max: "  
+Collections.max(IList));  
20. SList.add(new String("Ramesh"));  
21. SList.add(new String("Anand"));  
22. SList.add(new String("Gautam"));  
23. Collections.sort(SList);  
24. System.out.println("Ascending Order:  
" +SList);  
25. }  
26. }  
27. /* Output  
28. Ascending Order: [5, 7, 9]  
29. Index of Integer 9 in sorted list: 2  
30. Descending Order: [9, 7, 5]  
31. Shuffled List: [5, 9, 7]  
32. Min: 5  
33. Max: 9  
34. Ascending Order: [Anand, Gautam,  
Ramesh]  
35. */
```

---

<b>Line Nos. 5 &amp; 6:</b>	create two List objects: one of Integer type and the other of String type.
<b>Lines Nos. 8, 9, &amp; 10:</b>	populate the Integer list.
<b>Line No. 12:</b>	demonstrates how to apply sort algorithm on Integer list.
<b>Lines Nos. 14 &amp; 15:</b>	demonstrate binarysearch. Please note that binarysearch works only on sorted lists so we call it only after sorting the integer list.
<b>Lines Nos. 16 &amp; 18:</b>	demonstrate usage of shuffle and reverse algorithms.
<b>Lines Nos. 23–28:</b>	sort algorithm on string list.

## 24.8 Map Interface

Important features of Map interface are as follows:

- Map is similar to a dictionary.
- Each entry in the Map involves a pair of elements called keys and values. They are collectively referred to as key–value pairs.
- Each key maps to a particular value. For example, the roll number (key) maps to a name (value) in an attendance register.
- Map cannot contain duplicate keys, but can contain duplicate values.
- The association between key and value is one to one, i.e., each key should map to only one value.
- Collection framework provides two implementations of Map Interface: HashMap and TreeMap.

Table 24.5 describes the methods defined in the Map interface.

**Table 24.5** Map Interface description

Method	Remark
--------	--------

<code>V put (K key, V value)</code>	Create a new map for key and value
<code>V get (Object key)</code>	Returns the value associated with key
<code>V remove (Object key)</code>	Removes the mapping for key
<code>boolean contains Key (Object key)</code>	Checks if collection contains a mapping for key
<code>boolean contains Value (Object value)</code>	Checks if collection contains a mapping to value
<code>int size ()</code>	Returns the number of key–value pairs present in the collection
<code>boolean isEmpty ()</code>	Returns true if collection is empty
<code>void</code>	Clears all the mappings in the collection

<code>clear()</code>	
<code>Set&lt;K&gt; keySet()</code>	The method returns a Set which contains all the keys in the Map
<code>Collection&lt;V&gt; values()</code>	The method returns a Collection which contains all the values in the Map. The method returns Collection and not a Set as values can be duplicate whereas Keys are unique
<code>Set&lt;Map. Entry&lt;K, V&gt;&gt; entrySet ( )</code>	The method returns a Set which is populated with Map. Entry objects, key–value pairs can be obtained simultaneously by using methods defined in Map. Entry objects

## 24.9 Collection View of Map

It is not possible to attach an iterator directly to a Map. So in order to iterate over a Map, the Map interface provides methods which allow the Map to be viewed as a Collection. Example 24.5 demonstrates two such methods: `keySet()` and `values()`. Please note that Map stores only object references so it is not possible to use primitive data types like double or int. Instead we can use wrapper class such as Integer or Double.

## Example 24.5: Program to Demonstrate Collection View of Map

```
1. import java.util.Collection;
2. import java.util.Map;
3. import java.util.HashMap;
4. import java.util.Iterator;
5. import java.util.Set;
6. public class MapDemo{
7.     public static void main(String[]
args){
8.         Map<Object,String> map=new
HashMap<Object,String>();
9.         map.put(new Integer(120400),
"Ramesh");
10.        map.put(new Integer(120401),
"Anand");
11.        map.put(new Integer(120402),
"Gautam");
12.        System.out.println("Printing Keys:
");
13.        Set s = map.keySet();
14.        for(Iterator itr =
s.iterator();itr.hasNext();)
```

```

15. System.out.println(itr.next());
16. System.out.println("Printing Values:
");
17. Collection c = map.values();
18. for(Iterator itr =
c.iterator();itr.hasNext();)
19. System.out.println(itr.next());
20. }
21. }
22. /* Output
23. Printing Keys: 120401 120400 120402
24. Printing Values: Anand Ramesh Gautam
25. */

```

<b>Li ne No . 8:</b>	instantiates a Map object with Keys defined as Objects and Values defined as Strings. The Map uses the HashMap implementation.
<b>Li ne No s. 9, 10, &amp; 11:</b>	populate the Map with an Integer object and String.

<b>Line No.</b>	demonstrates how to use the <code>keySet ( )</code> method to get a Collection view of the keys. We use the Set Interface as Set cannot have duplicates like keys. Similarly, Line 19 shows how to get a collection view of Values.
-----------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Sometimes, we would like to modify the value associated with a key during iteration. Map interface provides a method `entrySet ( )` and a nested interface called `Map.Entry` to accomplish this. The method `entrySet ( )` returns a collection view of all the key–value pairs present in the Map. The nested interface `Map.Entry` is defined as follows:

```
public interface Entry {  
    K getKey(); V getValue(); V setValue(V  
    value);}
```

As seen from the above, the interface provides methods through which we can access and modify the Value associated with a key. Example 24.6 demonstrated how to use method `entrySet ( )` and `Map.Entry`



to iterate and access values associated with keys.

### **Example 24.6: Program to Demonstrate Collection View of Map**

```
1. import java.util.Map;
2. import java.util.HashMap;
3. import java.util.Set;
4. import java.util.Iterator;
5. public class MapDemo {
6.     public static void main(String[]
args) {
7.         Map<Object,String> map=new
HashMap<Object,String>();
8.         map.put(new Integer(120400),
"Ramesh");
9.         map.put(new Integer(120401),
"Anand");
10.        map.put(new Integer(120402),
"Gautam");
11.        Set s=map.entrySet();
12.        for(Iterator
it=s.iterator();it.hasNext();){
13.            Map.Entry m =
```

```

(Map.Entry) it.next();
14.    int rollno=(Integer)m.getKey();
15.    String name=(String)m.getValue();
16.    System.out.println("Roll No:" +
rollno + " Name:" +name );
17.    }
18.  }
19. }
20. /* Output
21. Roll No:120401 Name:Anand
22. Roll No:120400 Name:Ramesh
23. Roll No:120402 Name:Gautam
24. */

```

---

<b>L</b> <b>i</b> <b>n</b> <b>e</b>  <b>N</b> <b>O</b> <b>•</b> <b>1</b> <b>2</b> <b>:</b>	<p>creates a Set s that contains all the key–value pairs contained in the map. We associate an iterator to the Set in Line 13. Line 14 demonstrates how to create a Map.Entry object. Lines 15 and 16 show how to get the key and value using the Map.Entry methods. The for loop used in the program can be simplified using a for each loop as follows:</p>
--------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

---

```
for (Map.Entry< Object, String > e :  
map.entrySet()) {  
    System.out.println("Roll No:" +  
e.getKey() + " Name:" + e.getValue());  
}
```

---

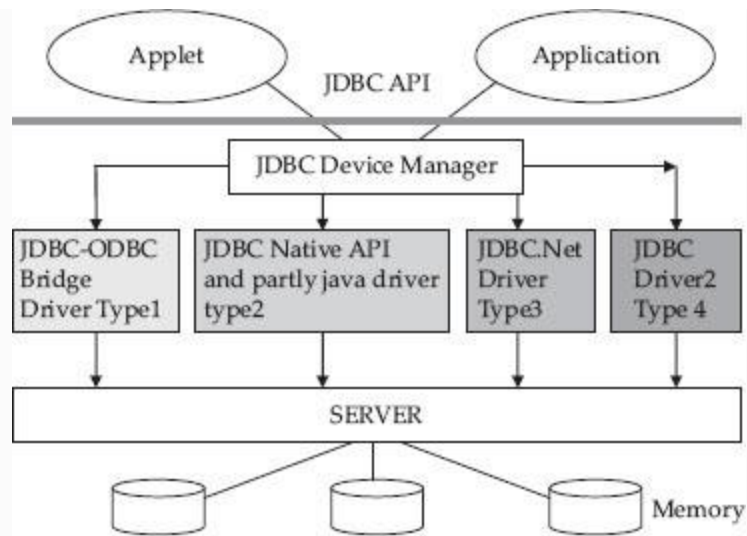
When using the above for loop line 12 `Set s=map.entrySet()` is also not required.

## 24.10 JDBC Database Connectivity

In order to connect to a database you will need Drivers. Firstly, as you will be requesting connection from Java program, you will need JDBC Manager to act as interface between Java components such as Applet or Application and Drivers.

Then to work with databases such as Oracle, MS Access, and MySQL which are third party database you will need drivers such as ODBC (Open Database Connectivity) which takes calls supplied by JDBC and converts them into formats suitable for proprietary databases such as MS Access and Oracle 10g, etc. This means that Database and ODBC must be available in

your machine. ODBC is shipped along with Windows OS. JDBC is a part of JDK which we have installed. Finally, MySQL also provides a connector `mysql-connector-java-5.1.3` and `mysql-5.1.51-win32.msi`. First download these databases as per your choice and install them. Paths are to be set by you in case it has not prompted you during installation. Procedure for setting path is provided in Section 16.6.2. Figure 24.4 describes this arrangement. Applet and application are from client side and Database is from the server side. The server side also can reside in the same machine for testing in which case its called local host.



**Figure 24.4** JDBC framework

**Type 1: JDBC-ODBC Driver:** Receives JDBC calls and hands over to ODBC driver. ODBC hands over these calls to manufacturers to library. This type is most widely used since ODBC drivers are available for almost all databases. This connector is a general connector and applicable to a wide variety of databases.

**Type 2:** Same as Type 1 but connector connects to a specific database. Hence, it is a specific connector. These are not popular since they are vendor specific.

**Type 3:** This is a net-based Data Connector in which JDBC converts the calls into middle tier server calls and, based on the actual server, the middle tier calls are converted to specific server. Obviously, as there is an additional conversion, this type is a slow connector.

**Type 4:** This is a pure Java driver that converts to vendor-specific database directly. It is machine independent since it is a pure Java product but the disadvantage is that you will need vendor-specific drivers.

## 24.11 Access Database Records Using MS Access Database

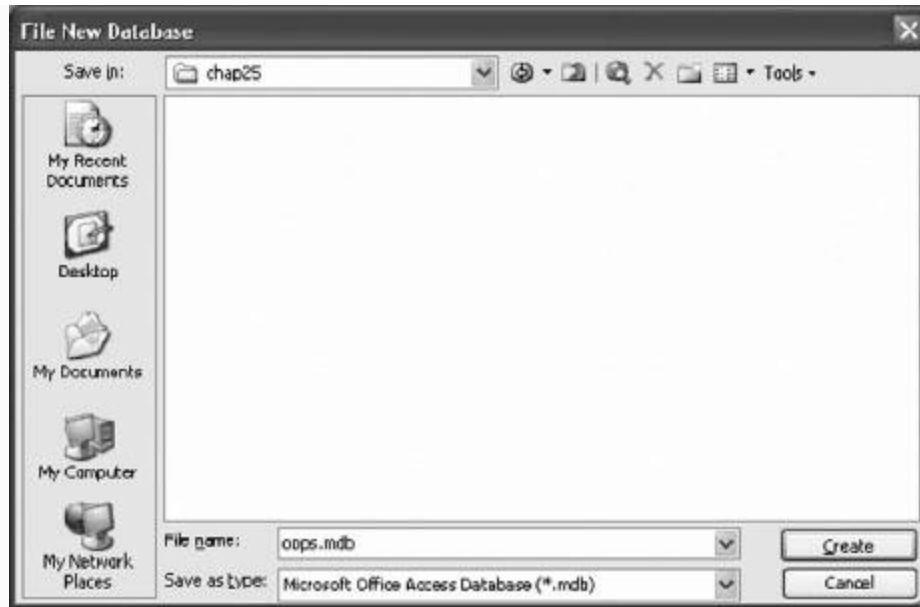
MS Access is the most popular and easily available database available along with MS office package. Students have easy access to MS Access and can gain knowledge of the process involved and easily migrate to Oracle and MySql. The application we will select is to create a database with fields idNum, Name, and Dept and Salary of Employees. Once the database is created, we will use Sql query to access the records and

display the result. There are three distinct phases for retrieving data from MS Access Database using JDBC-ODBC Bridge. They are:

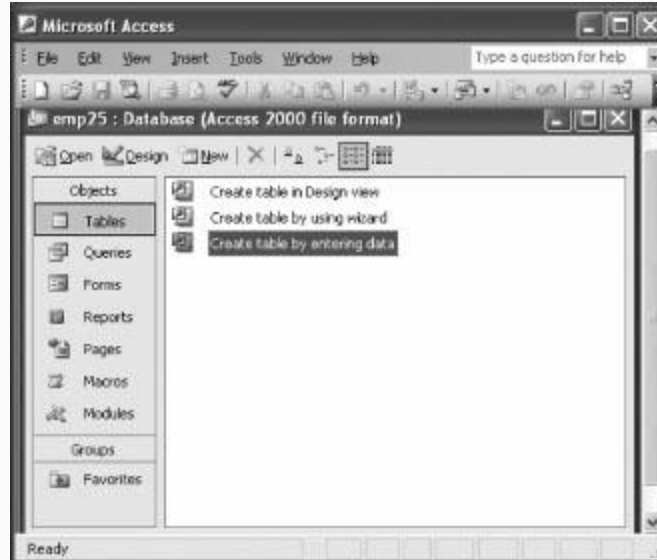
- Phase 1 : Create the MS Access database File
- Phase 2 : Create Data Source Name(DSN)
- Phase 3 : Write Java Application to access the MS Access database

## **Phase 1: Create MS Access database Table called oopsemp**

**Step 0:** Start ----→ MicroSoft Office----→ MS Access--→ create a new file--→Blank Database--→ enter filename to save as oops.mdb--→create in The Directory you are working. For example  
c:\oopsjava\workspace\oopstech2\com\oops\chap24

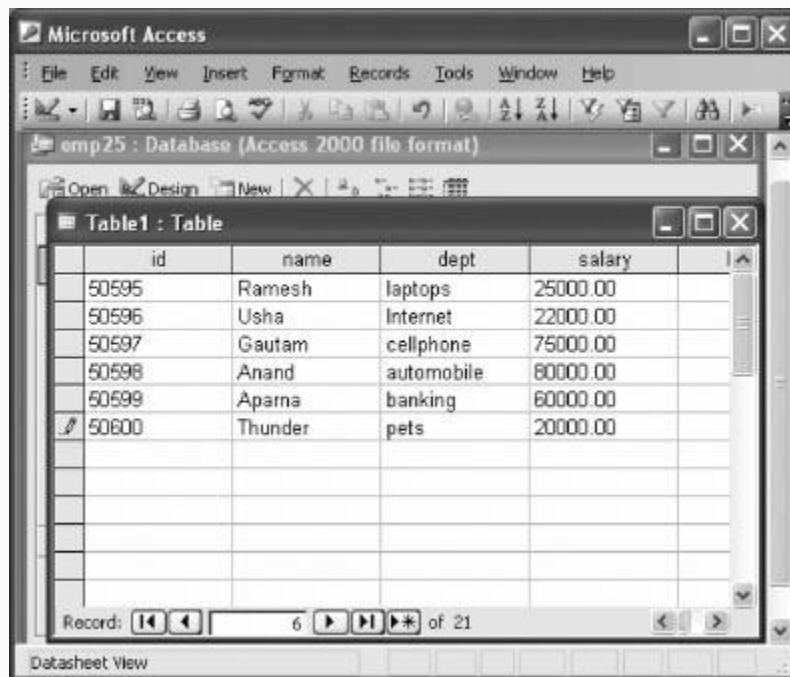


**Step 1:** Choose option 3 i.e. create table by entering data



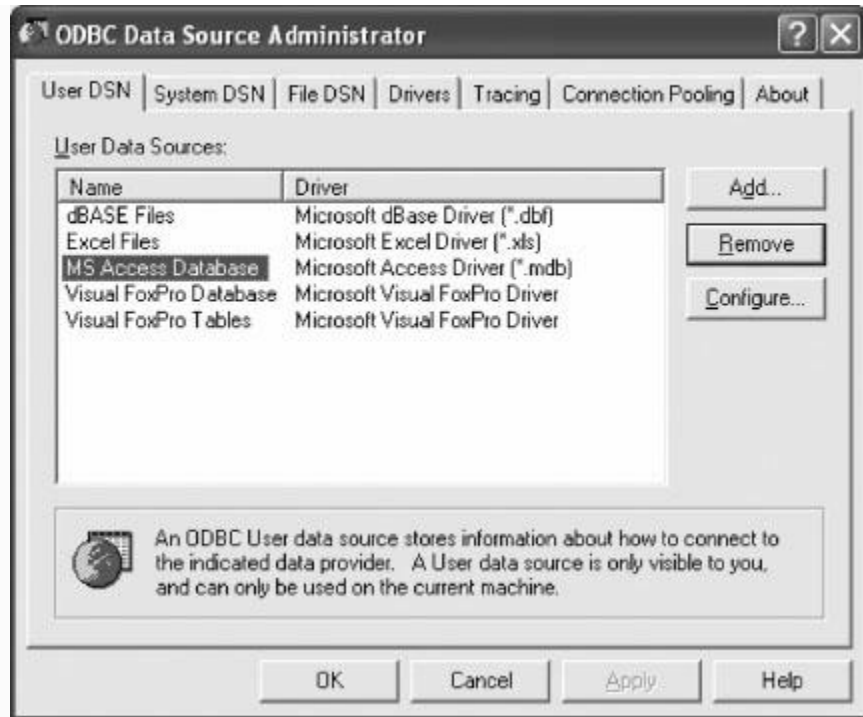
**Step 2:** Right click on field and select rename column and change field names to id, name, dept, salary. Then enter data in the relevant fields



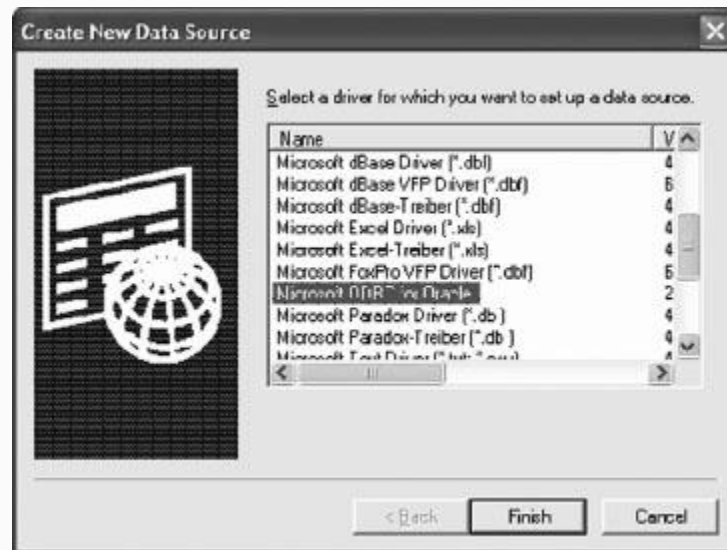


**Step 3:** File → save as → oopsemp24.mdb → ok. Click no for entering primary key

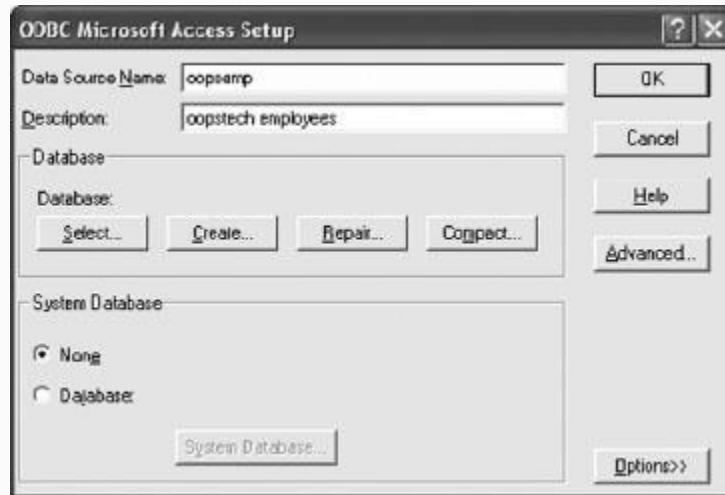
**Step 4:** Create Data Source Navigator DSN for JDBC ODBC connectivity Start → settings → control panel → Administrative Tools → Data Sources (ODBC) → MS Access DataBases → Add



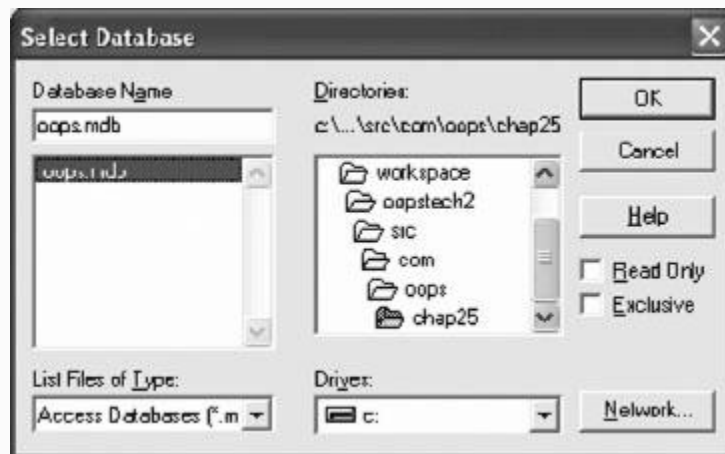
**Step 5:** Select the database for which we want to create DSN



**Step 6:** Enter data Source Name as oopsemp and description oopstech company employees. Press ok.



**Step 7:** Select the file from your directory, i.e., oops.mdb and press ok



**Step 8:** The control will return to ODBC Microsoft Access screen at Step 6. Press ok

**Step 9:** The control will return to ODBC Data Source Administrator screen at Step 4 → press ok.

DSN has been created. Now we are ready for phase 3, i.e., write a Java program.

## Phase 3: Write Java Program

We will show the methodology in steps

**Step 0:** `import java.sql.*;`

**Step 1:** Load and register the Driver by using  
`Class.forName()`

---

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

---

**Step 2:** Establish connection to a database

---

```
Connection con =  
DriverManager.getConnection  
  
("jdbc:odbc:oopsemp","","");  
                                oopsemp is the data  
source name we have selected while  
creating access table. User Name and  
Pass word field are empty strings.
```

---

**Step 3:** Create a statement : `Statement stmt =  
con.createStatement();`

**Step 4:** Execute the statement : `ResultSet rs =  
stmt.executeQuery("select * from oopsemp24");`

**Step 5:** Work on ResultSet

---

```
while (( rs.next()))  
{  
System.out.print(rs.getInt(reccount)+"
```

```
");  
  
System.out.print(rs.getString(reccount+1)  
)+ " ");  
  
System.out.print(rs.getString(reccount+2)  
)+" ";  
  
System.out.println(rs.getFloat(reccount+  
3)+" ");  
}
```

---

In our next example, we will put all the above steps into practice.

### **Example 24.7: AccessJDBCODBC.java A Program to Access Database File and Display the Fields**

---

```
1. package com.oops.chap24;  
2. import java.sql.*;  
3. public class AccessJDBCODBC {  
4. public static void main(String[]  
args) throws SQLException {  
5. int reccount=1;
```

```

6. //Register the Driver
7. try{
8.
Class.forName("sun.jdbc.odbc.JdbcOdbcDri
ver");
9. }catch ( Exception e )
{System.out.println("Unable to load
Driver");}
10.      Connection con =
DriverManager.getConnection("jdbc:odbc:o
opsemp","","");
11.      //create sql statement
12.      Statement stmt =
con.createStatement();
13.      ResultSet rs =
stmt.executeQuery("select * from
oopsemp24");
14.      System.out.println("\nid name
Dept Salary");
15.      while (( rs.next()))
16.
{System.out.print(rs.getInt(reccount)+"
");
17.
System.out.print(rs.getString(reccount+1
)+ " ");
18.
System.out.print(rs.getString(reccount+2
)+" ");
19.
System.out.println(rs.getFloat(reccount+
3)+" ");

```

```
20.      }  
21.      }  
22. }
```

---

---

```
Output: id name Salary  
id name Dept Salary  
50595 Ramesh laptops 24000.0  
50596 Usha internet 22000.0  
50597 Anand automobile 80000.0  
50598 gautam cellphones 75000.0
```

---

## 24.12 Access Database Records Using MySql Open Source Database

The reason for choosing MySQL is free and open source being provided by MySQL.com. For running the JDBC connectivity using MySQL, you will need `mysql-5.1.51-win32.msi` and `mysql-connector-java-5.1.13`. These are software for using mysql and connecting to databases through Java programs. Let us say that you have installed them in Directory `d:\Directory sql`. The path is

automatically set by installer. In this case also, we would follow a two-step strategy:

- Phase 1: Create the mysql database File
- Phase 2: Use JDBC connector and Write Java Application to access the MySql database

## Phase 1: Create the mysql database File

**Step 1:** To create database file : Start → programs → MySql Server 5.1 > MySql command line client → <enter> Enter password → mysql command prompt appears.

Database
-----
Information_schema
Mysql
oopsemp2
test

mysql> show databases; command shows databases available

To see what tables are available in oopsemp2 :

---

```
mysql> use oopsemp2; Database changed
mysql> show tables;
+-----+
| Tables_in_oopsemp2 |
+-----+
| oopsemp2           |
```



```
| oopsemp3 |  
+-----+  
2 rows in set (0.06 sec)
```

---

## To create a table called: oopsemp4 :

---

```
mysql> create table oopsemp4 ( id  
int,name char(10),dept char(10),salary  
float);  
Query OK, 0 rows affected (0.17 sec)
```

---

## To verify the field created: mysql> desc oopsemp4;

---

```
+-----+-----+-----+-----+-----+  
---+-----+  
| Field | Type | Null | Key | Default |  
Extra |  
+-----+-----+-----+-----+-----+  
---+-----+  
| id | int(11) | YES | | NULL | |  
| name | char(10) | YES | | NULL | |  
| dept | char(10) | YES | | NULL | |  
| salary | float | YES | | NULL | |  
+-----+-----+-----+-----+-----+  
---+-----+
```

---

## To enter data in to table oopsemp4

---

```
mysql> insert into oopsemp4 values
(50595,"Ramesh","laptops",20000.00);
Query OK, 1 row affected (0.09 sec).
Enter further records as required.
```

---

## To see the records we have just entered:

---

```
mysql> select * from oopsemp4;
+-----+-----+-----+-----+
| id | name | dept | salary |
+-----+-----+-----+-----+
| 50595 | Ramesh | Laptops | 20,000 |
| 50596 | Usha | Internet | 20,000 |
| 50597 | Anand | Automobiles | 80,000 |
| 50597 | Gautam | Cell phones | 80,000 |
+-----+-----+-----+-----+
4 rows in set (0.05 sec)
```

---

## To set a query: to see id and names of employees whose salary is more than 20000.00

---

```
mysql> select name from oopsemp4 where
salary >20000.00;
+-----+
| name |
```

```
+-----+
| Anand |
| Gautam|
+-----+
```

---

You have seen that we set queries and got the answers. Now will write a java program to connect to database and get the same answers.

**Phase 2:** Use JDBC connector and access the MySql database

### **Example 24.8: connect.java A Program to Access MySql Database File and Display**

```
1. package com.oops.chap24;
2. import java.sql.*;
3. public class Connect{
4. public static void main(String
args[]){
5. try{
6. String userName = "";
```

```

7. String password = "";
8. String url =
"jdbc:mysql://localhost/oopsemp2";
//database name
9. Class.forName
("com.mysql.jdbc.Driver").newInstance
();
10. Connection
conn=DriverManager.getConnection (url,
userName, password);
11. System.out.println ("Database
connection established");
12. Statement st =
conn.createStatement();
13. ResultSet rs=st.executeQuery("select
* from oopsemp3");// table name
14. System.out.println("\n output from
table :oopsemp3...");
15. System.out.print("-----
-----\n");
16. System.out.println("| Sur Name |
Last Name |");
17. System.out.println("-----
-----");
18. while(rs.next())
19. {
20. String username=rs.getString(1);
21. String password2=rs.getString(2);
22. System.out.print("\n| "+username);
23. System.out.print(" | "+password2);
24. System.out.print(" | ");
25. } // while

```

```

26. System.out.println("\n-----
-----");
27. //Handle oopsemp4 table
28. rs=st.executeQuery("select id,name
from oopsemp4 where salary>20000.0");//
table name
29. System.out.println("\n id & name for
salary>20000.00 from table :oopsemp4...");
30. System.out.print("-----
-----\n");
31. System.out.println("|Employee id no
| Name |");
32. System.out.println("-----
-----");
33. while(rs.next())
34. {int userid=rs.getInt(1);
35. String name =rs.getString(2);
36. System.out.print("\n| "+userid);
37. System.out.print(" | "+name);
38. System.out.print(" | ");
39. } // end of while
40. System.out.println("\n-----
-----");
41. } catch(Exception ex)
{ex.printStackTrace();}
42. } //main
43. } // class

```

Output : Database connection established  
output from table :oopsemp3...

```

-----
| Sur Name | Last Name |
-----

```

```

| ramesh | vasappanavara |
| Usha | vasappanavara |
| Anand | vasappanavara |
| Gautam | vasappanavara |
-----
id & name for salary>20000.00 from table
:oopsemp4...
-----
|Employee id no | Name |
-----
| 50597 | Anand |
| 50597 | Gautam |
-----

```

---

## 24.13 Access Database Records Using Oracle Database

There are three distinct phases for retrieving data from Oracle database using JDBC-ODBC Bridge. They are:

- Phase 1: Create the Oracle database File
- Phase 2: Create Data Source Name (DSN)
- Phase 3: Write Java Application to access the MS Access database

### **Phase 1:** Create Oracle database file

We are assuming that you have Oracle installed in your computer system and SQL>prompt is available. We will show only

in command prompt usage, while web-based approach is also allowed by Oracle 10 and later versions. All Oracle databases file in Oracle 10g and later will have User name: system and password: oracle. Other older oracle versions use username: scott and password: tiger. Get SQL prompt on the screen.

---

```
SQL> connect system; Enter
password:oracle<enter> Connected. SQL>
SQL> select * from cat ; will list
out all tables in system
-----
-----
PRODUCT_USER_PROFILE SYNONYM
HELP TABLE
OOPSEMP5 TABLE
179 rows selected.
SQL>
SQL> SQL> create table oopsemp6( idNum
int, name varchar2(10),dept
varchar2(10),salary float);
Table created.
OOPSEMP6 TABLE
OOPSEMP5 TABLE
180 rows selected.
SQL> desc oopsemp6; will show the field
in table oopsemp6
Name Null? Type
```

```

-----
-----
IDNUM NUMBER(38)
NAME VARCHAR2(10)
DEPT VARCHAR2(10)
SALARY FLOAT(126)
SQL> insert into oopsemp6
values(&idNum,'&name','&dept',&salary);
Enter value for idnum: 50595
Enter value for name: Ramesh
Enter value for dept: laptops
Enter value for salary: 20000.00
old 1: insert into oopsemp6
values(&idNum,'&name','&dept',&salary)
new 1: insert into oopsemp6
values(50595,'Ramesh','laptops',20000.00
)
1 row created.
We can create further records.
SQL > SELECT * FROM OOPSEMP6; will
show you the data we have entered
IDNUM NAME DEPT SALARY
-----
---
50595 Ramesh laptops 20000
50596 Usha Internet 22000
Now we are ready to proceed to Phase

```



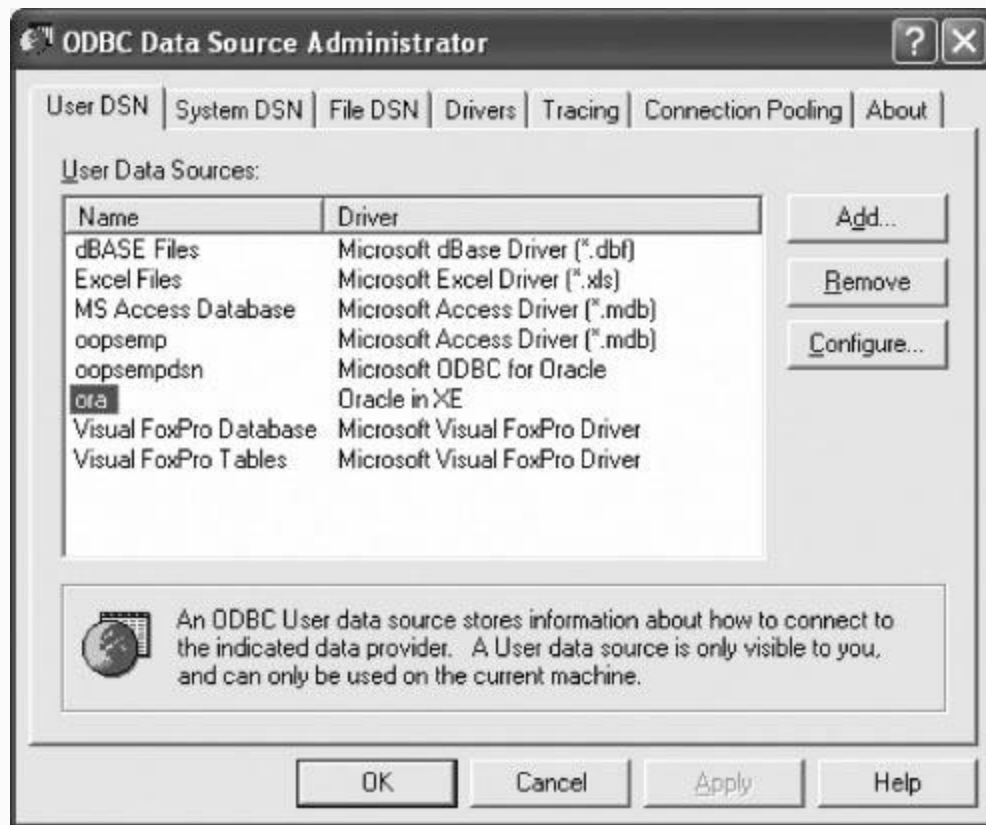
## Phase 2: Create a Data Source Name (DSN)

**Step 1:** Create Data Source Navigator DSN for JDBC ODBC connectivity

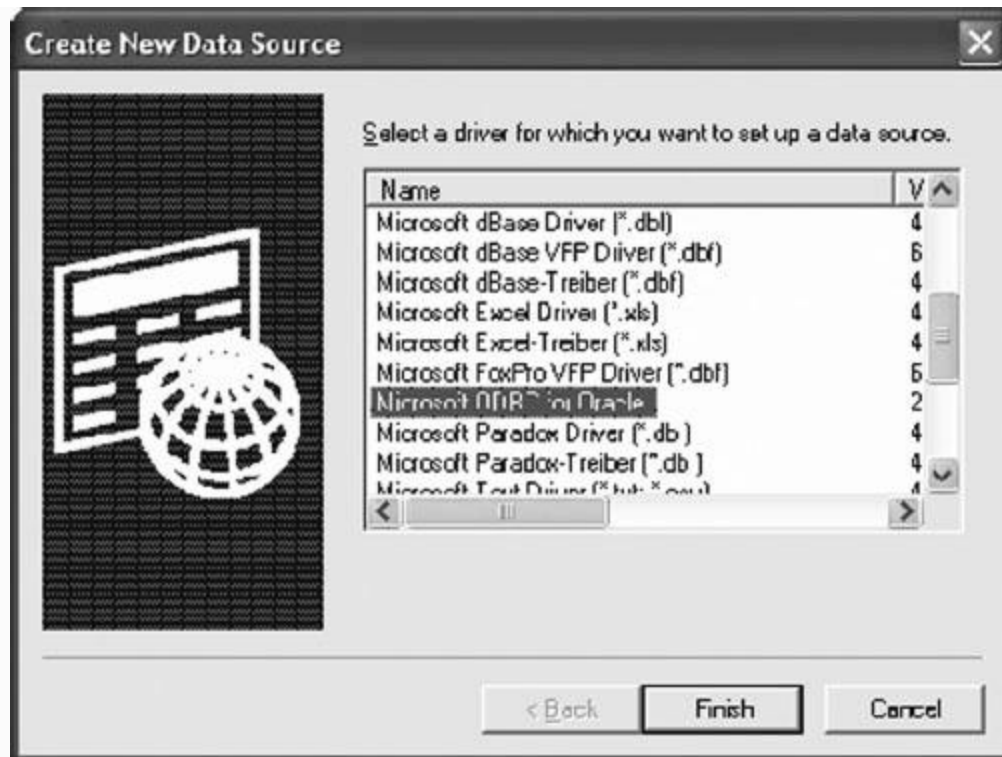
---

Start → settings → control panel → Administrative Tools → Data Sources (ODBC) → Add

---



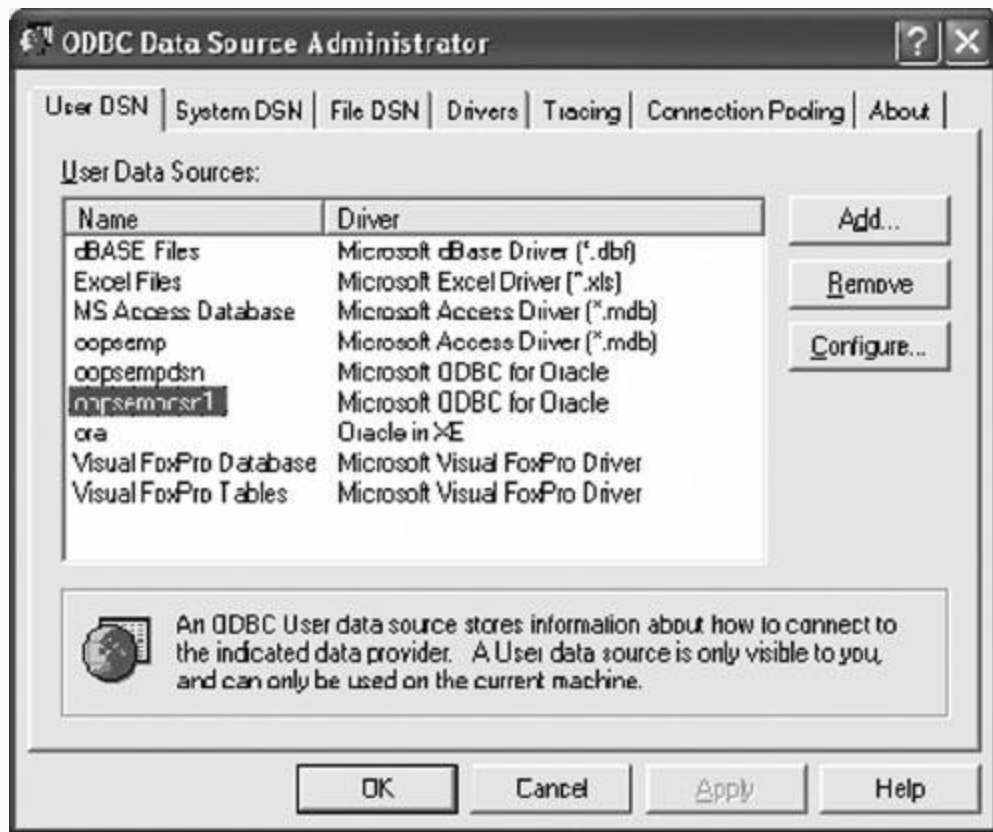
**Step 2:** Select the database for which we want to create DSN-Press Finish.



**Step 3:** Enter data Source Name as oopsempdsn and description oops employees. Description system and Press ok.



**Step 4:** The control will return to ODBC Data Source Administrator screen at Step 1 → you can see oopsempdsn1 listed →press ok.



DSN has been created. Now we are ready for phase 3, i.e., write a Java program.

**Phase 3:** Write Java Application to access the Oracle database

**Example 24.9:** AccessOracle.java A program to Access Oracle Database

---

```
1. package com.oops.chap24;
2. import java.sql.*;
3. public class accessoracle{
4. public static void main(String
args[]){
5. try{
6. String userName = "system";
7. String password = "oracle";
8. String url = "jdbc:odbc:oopsempdsn";
//database name
9. Class.forName
("sun.jdbc.odbc.JdbcOdbcDriver");
10. Connection
conn=DriverManager.getConnection
(url,userName,password);
11. System.out.println ("Database
connection established");
12. Statement st =
conn.createStatement();
13. //Handle oopsemp5 table
14. ResultSet rs=st.executeQuery("select
idNo ,name from oopsemp5 where
15. salary>20000.0");// table name
16. System.out.println("\n idNo & name
for salary>20000.00 from table
:oopsemp5...");
17. System.out.print("-----
-----\n");
18. System.out.println("|Employee id no
| Name |");
19. System.out.println("-----"
```

```

-----");
20. while(rs.next())
21. { int userid=rs.getInt(1);
22. String name =rs.getString(2);
23. System.out.print("\n| "+userid);
24. System.out.print(" | "+name);
25. System.out.print(" | ");
26. } // end of while
27. System.out.println("\n-----
-----");
28. } catch(Exception ex)
{ex.printStackTrace();}
29. } //main
30. } // class
Output : Database connection established
idNo & name for salary>20000.00 from
table :oopsemp5...
-----
|Employee id no | Name |
-----
| 50596 | Usha |
| 50597 | Anand |
| 50599 | Gautam |
-----

```

---

## Example 24.10: Inserting Records into a Database

```
1. package com.oops.chap24;
2. import java.sql.*;
3. public class InsertDB{
4. public static void main(String
args[]){
5. try{String userName = "system";
6. String password = "oracle";
7. String url = "jdbc:odbc:oopsempdsn";
//database name
8. Class.forName
("sun.jdbc.odbc.JdbcOdbcDriver");
9. Connection conn =
DriverManager.getConnection (url,
userName, password);
10. System.out.println ("Database
connection established");
11. Statement st =
conn.createStatement();
12. //Handle oopsemp5 table
13. System.out.println("\n Inserting a
record in to database");
14. st.executeUpdate("insert into
oopsemp5
values(50600,'Prahlad','Toys',40000.00)"
);
15. ResultSet rs=st.executeQuery("select
idNo ,name from oopsemp5");// table name
16. System.out.println("\n idNo & name
from table :oopsemp5...");
```

```

17. System.out.print("-----
-----\n");
18. System.out.println("|Employee id no
| Name |");
19. System.out.println("-----
-----");
20. while(rs.next())
21. {int userid=rs.getInt(1);
22. String name =rs.getString(2);
23. System.out.print("\n| "+userid);
24. System.out.print(" | "+name);
25. System.out.print(" | ");
26. } // end of while
27. System.out.println("\n-----
-----");
28. } catch(Exception ex)
{ex.printStackTrace();}
29. } //main
30. } // class

```

Output : Database connection established  
Inserting a record in to database  
idNo & name from table :oopsemp5...

```

-----
|Employee id no | Name |
-----
| 550595 | ramesh |
| 50596 | Usha |
| 50597 | Anand |
| 50599 | Gautam |
| 50600 | Prahlad |
-----

```

---

## Example 24.11: Updating a Field in a Records in a Database

```
package com.oops.chap24;
import java.sql.*;
public class UpdateDB{
public static void main(String args[]){
try{
String userName = "system";
String password = "oracle";
String url = "jdbc:odbc:oopsempdsn";
//database name
Class.forName
("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn =
DriverManager.getConnection (url,
userName, password);
System.out.println ("Database connection
established");
Statement st = conn.createStatement();
//Handle oopsemp5 table
System.out.println("\n Updating a record
in to database");
st.executeUpdate("update oopsemp5 set
salary = 35000 where idNo=50595");
```



```

ResultSet rs=st.executeQuery("select
idNo ,name, salary from oopsemp5");//
table name
System.out.println("\n idNo&name&salary
from table :oopsemp5...");
System.out.print("-----
-----\n");
System.out.println("|Employee id no |
Name | Salary |");
System.out.println("-----
-----");
while(rs.next())
{int userid=rs.getInt(1);
String name =rs.getString(2);
float sal = rs.getFloat(3);
System.out.print("\n| "+userid);
System.out.print(" | "+name);
System.out.print(" | "+ sal);
} // end of while
System.out.println("\n-----
-----");
} catch(Exception ex)
{ex.printStackTrace();}
} //main
} // class

```

---



---

**Output:** Database connection established  
Updating a record in to database  
idNo & name from table :oopsemp5...

```

-----
|Emp idno | Name | Salary |
-----
| 50595 | ramesh | 35000.0 |
| 50596 | Usha | 22000.0 |
| 50597 | Anand | 80000.0 |
| 50599 | Gautam | 85000.0 |
| 50600 | Prahlad | 40000.0 |
-----

```

## Example 24.12: Deletion Database Record

```

package com.oops.chap24;
import java.sql.*;
public class DeleteDB{
    public static void main(String args[])
    {
        try{String userName = "system";
        String password = "oracle";
        String url = "jdbc:odbc:oopsempdsn";
        //database name
        Class.forName
        ("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection conn =

```

```
DriverManager.getConnection (url,
userName, password);
System.out.println ("Database connection
established");
Statement st = conn.createStatement();
//Handle oopsemp5 table
System.out.println("\n Deleting records
to database");
st.executeUpdate("delete oopsemp5 where
idNo=50600");
} catch (Exception ex)
{ex.printStackTrace();}
}
}
```

---

You can use module presented at Ex1 to view the database records. You can observe that records with idNo=50600 have been deleted.

## 24.14 Prepared Statement and Callable Statements

We have used Statement Object: Statement st = conn.createStatement(); where conn is an object : Connection conn = DriverManager.getConnection (url, userName, password);

Finally we have used st :  
st.executeUpdate("delete oopsemp5 where  
idNo=50600");

### *24.14.1 PreparedStatement*

Statement carries out parsing every time statement gets executed. So far there are no repetitions of statements. What is if there is loop in which a statement gets executed several times thus wasting compiler time.

PreparedStatement, on the other hand, is a pre-parsed and compiled statement that needs not be compiled every time a statement gets executed. Thus, it is beneficial to use PreparedStatement if multiple executions are expected.

---

```
//Handle oopsemp5 table
PreparedStatement st =
conn.prepareStatement ("update oopsemp5
set salary=? where idNo=?");
System.out.println("\n Updating a
record in to database");
st.setFloat(1,50000.00f); // applies
to first ? mark
st.setInt(2,50595); // applies to
second ?
```

```
st.executeUpdate("update oopsemp5 set  
salary = 35000 where idNo=50595");
```

---

### **Example 24.13: PreparedDB.java A Program to Show the Usage of PreparedStatement**

```
1. package com.oops.chap24;  
2. import java.sql.*;  
3. public class PreparedDB{  
4.     public static void main(String  
args[]){  
5.     try{  
6.         a. String userName = "system";  
7.     String password = "oracle";  
8.     String url = "jdbc:odbc:oopsempdsn";  
//database name  
9.     Class.forName  
("sun.jdbc.odbc.JdbcOdbcDriver");  
10.    Connection conn =  
DriverManager.getConnection (url,  
userName, password);  
11.    System.out.println ("Database  
connection established");
```

```

12. //Handle oopsemp5 table
13. System.out.println("Updating a
record in to database");
14. PreparedStatement st =
conn.prepareStatement("update oopsemp5
set salary = ? where idNo = ?");
15. st.setFloat(1,50000); // applies to
first ? mark
16. st.setInt(2,50595); // applies to
second ?
17. st.executeUpdate();
18. Statement st1 =
conn.createStatement();
19. ResultSet
rs=st1.executeQuery("select idNo ,name
,salary from oopsemp5");// table name
20. System.out.println("idNo & name &
salary :oopsemp5...");
21. System.out.print("-----
-----\n");
22. System.out.println("|Employee id no
| Name |Salary |");
23. System.out.println("-----
-----");
24. while(rs.next())
25. {
26. int userid=rs.getInt(1);
27. String name =rs.getString(2);
28. float sal = rs.getFloat(3);
29. System.out.print("\n| "+userid);
30. System.out.print(" | "+name);
31. System.out.print(" | "+ sal);

```

```
32. } // end of while
33. System.out.println("\n-----
-----");
34. } catch(Exception ex)
{ex.printStackTrace();}
35. } //main
36. } // class
```

---

**Output:** Database connection established  
Updating a record in to database  
idNo & name & salary from table  
:oopsemp5...

```
-----
|Employee id no | Name | Salary |
-----
| 50595 | ramesh | 50000.0
| 50596 | Usha | 22000.0
| 50597 | Anand | 80000.0
| 50599 | Gautam | 85000.0
```

---

### *24.14.2 Callable Statements*

Callable statements are statements in a Java program that call a stored procedures and functions. Stored procedures are a set of statements executed on database and the result is sent to calling Java program.

**Step 0:** We have to create a stored procedure say sutti.sql. Go to sql prompt and enter

---

```
SQL> select * from oopsemp7;
IDNO NAME SALARY
-----
50595 Ramesh 20,000
50596 Usha 22,000
```

---

**Step 1:** Enter at SQL Prompt

---

```
SQL> create or replace procedure
sutti( no in int, pay out float) as sal
float;
2 begin
3 select salary into sal from
oopsemp7 where idno=no;
4 pay:=sal+10000;
5 end;
6 / press enter
Procedure created.
SQL>
```

---

Alternatively, you can open notebook anywhere, enter the program, and save it as sutti.sql. The contents can be :

---

```
create or replace procedure sutt11(no in
int, pay out float) as sal float;
```



```
begin
select salary into sal from oopsemp5
where idNo=no;
pay := sal + 10000; // observe it is :=
end;
/ // after pasting code in SQL prompt
press enter
Copy the content of file using copy and
paste it on to SQL> prompt line. It will
appear as
SQL> create or replace procedure suttil
2 (no in int, pay out float) as sal
float;
3 begin
4 select salary into sal from oopsemp5
where idNo=no;
5 pay := sal + 10000;
6 end;
7 /
```

***Press Enter. You will see procedure created message on the sql prompt screen. You are ready.***

---

## **Example 24.14: StoredDB.java A Program to Show the Usage of CallableStatement**

---

```
package com.oops.chap24;
import java.sql.*;
public class StoredDB{
    public static void main(String args[])
    {
try{String userName = "system";
String password = "oracle";
String url = "jdbc:odbc:oopsempdsn";
//database name
Class.forName
("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn =
DriverManager.getConnection (url,
userName, password);
System.out.println ("Database connection
established");
//Handle oopsemp5 table
System.out.println("Using a callable
statement");
CallableStatement st2 =
conn.prepareCall("{ call sutt1(?,?)}");
st2.setInt(1,50596); // applies to first
? mark
//Register the output as float data type
st2.registerOutParameter(2,Types.FLOAT);
st2.execute();
float incrPay = st2.getFloat(2);
    System.out.println("\n enhanced pay " +
incrPay);
```

```
conn.close();
} catch(Exception ex)
{ex.printStackTrace();}
} //main
} // class
Output : Database connection established
Using a callable statement
enhanced pay 32000.0
```

---

## **How to update fields in a record of database?**

In the above example , the stored procedure did not update the record; instead it returned up dated salary to calling procedure. What should we do in case we need to update the database. Ex 4 at the end of the chapter gives the program.

### **24.15 Servlets**

We are aware that applets are dynamic and that they reside embedded on to web browser and can extend the functionality of the web browser across the Internet. Servlets like applets are small server side programs that reside on the web servers and

extend the web servers functionality across the Internet. Hence, the name *Servlets*. As an example, consider a server is hosting an application for centralized inventory control system for its multi-country multi-location branches. The company provides information about Inventory on real time through web pages to all its branch offices. The information must reflect the latest position of the stock in the database. Servlets offer best possible means to extend the web server to client and in addition provide advantages such as platform independency, ability to interact with applets, databases and sockets, etc.

## **What environment we need to put Servlets to work?**

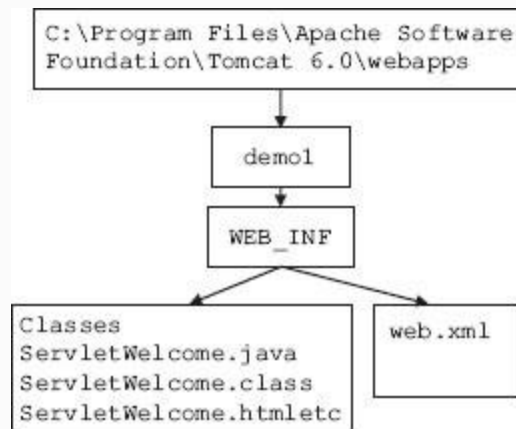
To host a servlet you need a server. The best server widely used in industry is Tomcat Apache Server. This software can be downloaded :[Jakarta.apache.org](http://Jakarta.apache.org) and can be downloaded to: C:\Program Files\ Apache Soft Foundation\Tomcat 5.0. To start the server go to bin directory and click on

C:\Program Files\Apache Software Foundation\Tomcat 6.0\bin> tomcat6.exe.  
You will also be able to start the tomcat server by using start→programs→Tomcat6.0→tomcat6 or by permanently setting tomcat server to be worked each time you switch on the system by selecting tomcat6 by running config command from start→run→config

**Servlet libraries-servlet-api.jar:** Note that Servlets is not part of pure java. Hence, you need to include servlet-api.jar which is present c:\Program Files\Apache Software Foundation\Tomcat 6.0\common\lib\server.jar file as part of class path. The procedure for doing so is: start→ settings→controlpanel→system→advanced→ environment variables→ select class path from bottom→ edit → enter the class path shown above → press ok

**S** Write servlet code ServletWelcome.java in  
**t** the directory called Demo1 and place the  
**e** directory Demo1 C:\Program Files\Apache  
**P** Software Foundation\Tomcat

- 6.0\webapps. The content of directory Demo1
- should be as shown in Figure 24.5.



**Figure 24.5** Where to house Servlet programs?–Organization

**Example 24.15:** ServletWelcome.java  
A program to Show the Usage of  
Servlet

---

```
1. import java.io.*;
2. import javax.servlet.*;
3. public class ServletWelcome extends
GenericServlet {
4. public void service(ServletRequest
req,ServletResponse res) throws i.
ServletException,IOException{
5. res.setContentType("text/html");
6. PrintWriter pw = res.getWriter();
7. pw.println("<B>Hello Students!
Welcome to Servlet Chapter");
8. }
9. }
```

---

**Compile** `javac ServletWelcome.java`  
**into** `C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\demo1\WEB-INF\classes`. **Note** that loading class file into the specified directory is mandatory for Tomcat to pick up the class file.

**Step 1:** Add servlet name and mapping to `web.xml` file in the following `C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\demo1\WEB-`

INF.Web.xml file can be opened by notepad or wordpad for editing. Servlets are embedded onto web.xml file. A typical web.xml file would look like:

### **Example 24.16: A typical web.xml File for ServletDemo Servlet**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD
  Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-
  app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-
name>ServletWelcome</servlet-name>
    <servlet-
class>ServletWelcome</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-
name>ServletWelcome</servlet-name>
```



```
<url-pattern>/ServletWelcome</url-  
pattern>  
</servlet-mapping>  
</web-app>
```

---

**Step 2:** Start the tomcat server

**Step 3:** Send request on web browser

---

```
http://localhost:8080/demo1/ServletWelco  
me
```

---

## Output: Hello Students! Welcome to Servlet Chapter

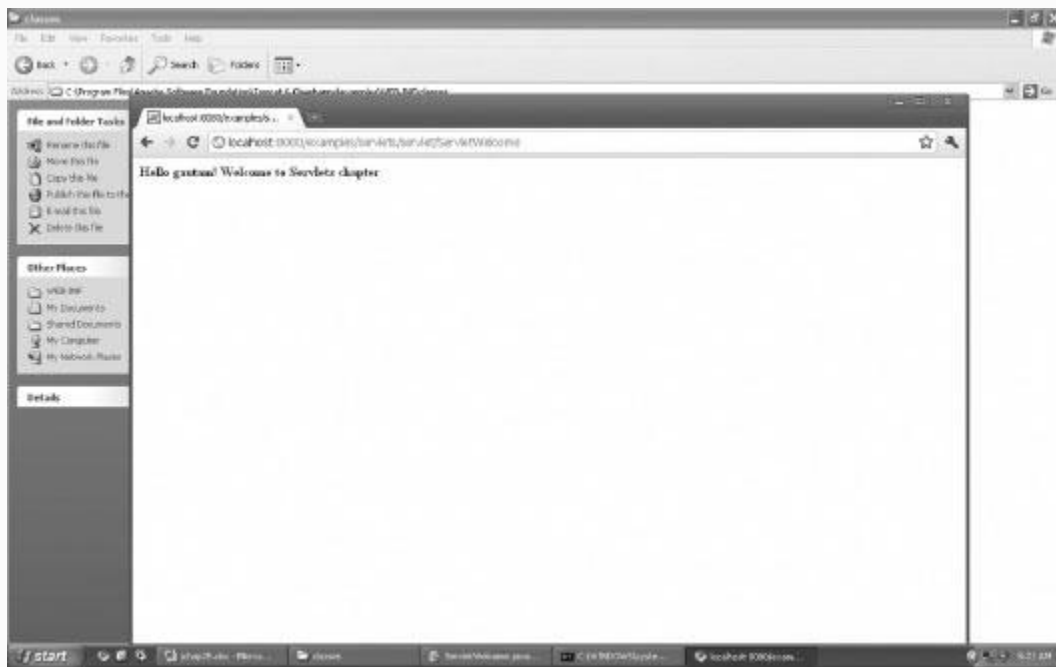
<b>L i n e N o . : 2</b>	imports javax.servlet.* ; a package essential to run servlet programs. Make sure you have included Servlets-api.jar file in your class path.
----------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------

<b>L i n</b>	creates a class ServletWelcome as subclass of GenericServlet. That provides init(), destroy() methods. We will override
----------------------	-------------------------------------------------------------------------------------------------------------------------

**e** `service()` method for our own  
**N** implementation. `Service()` method takes care of  
**o** clients requests by accepting `ServletRequest`  
**.** object `res` and gives out `ServletResponse` object  
**3** `res`.  
**:**

**L** sets the content type to `html/text`. It is an  
**i** indication that browser to interpret the data in  
**n** `html`.  
**e**  
**N**  
**o**  
**.**  
**5**  
**:**

**L** use `getWriter()` that obtains `PrintWriter`  
**i** object `pw`. In Line No. 7, the object `pw`  
**n** writes the string to web browser to display the  
**e** message.  
**N**  
**o**  
**s**  
**.**  
**6**  
**&**  
**7**  
**:**



### *24.15.1 Servlets API and javax.servlet Package*

Servlet API contains `javax.servlet` and `javax.servlet.http`. These are part of Tomcat `wservlet-api.jar` and not part of J2SE 5. `javax.servlet` package supplies all the interfaces required for the implementation and is shown in [Table 24.6](#), while the important classes in the package are shown at [Table 24.7](#). Interfaces are shown in [Table 24.8](#).

**Table 24.6** Interfaces provided javax.servlet package

Javax.servlet package interfaces	Remarks
Servlet	Life cycle methods of servlets included
ServletRequest	Gets data from the client request
ServletResponse	Sends data to client request
ServletConfig	Initial params are set
ServletContext	Servlets can log events & can get environment info

**Table 24.7** Classes provided javx.servlet package

Javax.servlet package Classes		Remarks
GenericServelet	Implements servlets * config interfaces	
ServletInputStream	InputStreamfor the client request	
ServletOutputStream	OutputStream for sending data	
ServletException	Servlet error	
ServletUnavailableEx ception	Servlet unavailable	

**Table 24.8** Servlet interface–Methods

Javax.servlet interface Methods		Remarks
void service (ServiceRequest req, ServiceResonse res) throws Sevlet Exception, IOException		Attends request from client and sends the response
void init( ServletConfig config)		Initialization params are set
ServletConfig getServletConfig()		Servlets config information & ServletsContext for accessing environment params
String getServletInfo()		Returns servlet author 7version
void destroy()		Called when servlet is unloaded
ServletException		Servlet error
ServletUnavailableExcepti on		Servlet unavailable

ServletRequest and ServletResponse interphases shown above receive the request object from client and prepares response to be sent. Servlet packages define two types of

abstract classes. One is `GenericServlet` belonging to `javax.servlet` and the other is `HttpServlet` belonging to package `javax.servlet.http`. In our next example, we will show a program extending to `GenericServlet`. In this example, we would read the parameter names and values contained in clients request by copying them into object `enum` of `Enumeration` class of `Java's Utility Package`.

### **Example 24.16: PostParam.java A Program to Show Extending to GenericServlet.**

```
1. import java.io.*;
2. import java.util.*;
3. import javax.servlet.*;
4. public class PostParamServlet extends
GenericServlet {
5. public void service(ServletRequest
req , ServletResponse res)
6. throws ServletException, IOException{
7. // print writer
```

```

8. PrintWriter pwtr = res.getWriter();
9. Enumeration enm =
req.getParameterNames();
10. // Display Parameter names and
values
11. while( enm.hasMoreElements() ){
12. String stg = (String)
enm.nextElement();
13. pwtr.print(stg + " = ");
14. String pval= req.getParameter(stg);
15. pwtr.println(pval);
16. }
17. pwtr.close();
18. }
19. }

```

---

**Step1: Compile javac PostParamServlet.java into C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\demo2\WEB-INF\classes. Load the html file PostParamServlet.html also in the same directory.**

**Li  
ne  
N**

shows that PostParamServlet extends to GenericServlet class.



<b>o. 5:</b>	
<b>Li ne N o. 9:</b>	creates a <code>PrintWriter</code> Object <code>pwtr</code> to send data to client
<b>Li ne N o. 10 :</b>	Enumeration <code>enm = req.getParameterNames()</code> ; pushes the names and values contained in client message into enumeration object <code>enm</code> . Using Line No. 12: <code>while( enm.hasMoreElements()) {</code>
<b>Li ne N os . 14 &amp; 15 :</b>	send name and roll number to client through <code>pwtr</code> object

#### HTML File: `PostParamServlet`

```
<html>
```

```
<body>
```

```
<center>
<form name="Form1"
method ="post"
action
="http://localhost:8080/demo2/PostParamS
ervlet">
    <table>
<tr>
    <td><B>StudentName</td>
<td><input type = textbox
name="studentname" size="24" value ="">
</td>
</tr>
<tr>
    <td><B>RollNumBer</td>
    <td><input type=textbox
name="studentnumber" size="24" value
=""></td>
</tr>
</table>
<input type="submit" value = "Submit">
</body>
</html>
```

Web.xml: As shown in Example 24.16b

---

**Step 2:** Start the Tomcat Server

**Step 3:** Send request on web browser

---

http://localhost:8080/demo2/PostParamSer  
vlet

---

StudentName:ShraddaPandy

RollNumBer: 50596

Submit

---

OUTPUT from server

studentname = ShraddaPandey

studentnumber = 50596

---

## *24.15.2 HttpServletRequest*

Out of the two abstract classes, namely, `GenericServlet` and `HttpServlet`, `HttpServlet` provides better interface and is widely used. The important methods are:

- `Service()` receives `ServletRequest` and `ServletResponse` object that provide access to stream inputs of type character based or byte based.
- Service requests for `HttpServlet` are get and post. Get request gets the information from the server and post sends data to a server such as information from form or authentication details, etc. `HttpServlet` defines `doGet()` and `doPost()` methods which are called by `service()` method. Interfaces `HttpServletRequest` and `HttpServletResponse` handle `doGet()` and `doPost()` and organize client server interaction.

Whenever `doGet()` or `doPost()` methods are called, object of `HttpServletRequest` is created by the server and passed on to `service()` method as an argument. The `service()` method executes the service request and makes the response ready for dispatch to client. A few of the important methods supported by `HttpServletRequest` interface are shown in Table 24.9.

**Table 24.9** `HttpServletRequest` interface

HttpServletRequest interface	Remarks
String getName()	Gets the argument sent to servlet by get or post requests
Enumeration getParameterNames()	Returns names of all parameters
String[] getParameterValues( String names)	Names of all parameters sent to servlet
Cookie[] getCookies()	Return an array of cookie object
HttpSession getSession(boolean create)	Returns current session of the client. Helps in unique identification of the client

### 24.15.3 *HttpServletResponse*

The main job of `HttpServletResponse` is to prepare the response to be sent to the client on receipt of `HttpServletRequest` Object. The web servers that is executing the servlet makes an object of `HttpServletResponse` and hands it over to `service()` method. A few of the

important methods are presented in Table 24.10.

**Table 24.10** HttpServletRequest methods interface

HttpServletRequest interface		Remarks
<code>ServletOutputStream getOutputStream()</code>		Gets a Byte-based output stream to be sent to client
<code>PrintWriter getWriter()</code>		Gets a character-based output stream for sending text data
<code>void setContentType</code>		Sets Multipurpose Internet Mail Extensions (MIME) type. Ex “text/html”
<code>void addCookie(cookie cookie)</code>		Returns an array of cookie object
<code>HttpSession getSession(boolean create)</code>		Returns current session of the client. Helps in unique identification of the client

### *24.15.4 HttpServlet Class—Get Request with Data from Client*

HttpServlet class provides several methods to perform tasks of receiving client request and sending servers response. The important methods are doGet () and doPost (). The other methods include delete (), doPut () and, of course, service () method. These methods all handle HTTP functions. In our next example, we will show Http Get Request. In this model, we present a form for the user to select his favourite subject by offering options and a submit button. The action statement in the html file links the servlet to be called by the server. Server calls the prescribed servlet and prepares the response and sends them to client. The client selects a preferred subject and submits the choice to servlet at server. The Http request is received by the servlet and prepares the response.

**Example 24.18:**

**SubjectGetServlet.java A Program to Show Usage of HTTP Get Request**

## Step 1: Create HTML File: SubjectGetServlet.html in the Directory C:\PROGRA~1\APACHE~1\TOMCAT~1.0\ webapps\Demo3\WEB-INF\classes>

---

```
1. <html>
2. <body>
3. <center>
4. <form name="Form1"
5.
   action="http://localhost:8080/Demo3/SubjectGetServlet">
6. <B>Subject:</B>
7. <select name="subject" size="1">
8. <option
   value="Maths">Mathematics</option>
9. <option
   value="Physics">Physics</option>
10.    <option
   value="Chemistry">Chemistry</option>
11.    </select>
12.    <br><br>
13.    <input type=submit
   value="Submit">
14.    </form>
15. </body>
16. </html>
```

---



Subject:

DropDown List. User selects any one subject from options at Line Nos. 7, 8, and 9.

**Step 2:** Create Servlet program

SubjectGetServlet.java in the directory:

C:\PROGRA~1\APACHE~1\TOMCAT~1.0\webapps\Demo3\WEB-INF\classes>

---

```
1. import java.io.*;
2. import javax.servlet.*;
3. import javax.servlet.http.*;
4. public class SubjectGetServlet
   extends HttpServlet {
5.     public void doGet(HttpServletRequest req,HttpServletResponse res)
6.     throws ServletException, IOException
7.     {
8.         String
subject=req.getParameter("subject");
9.         res.setContentType("text/html");
10.        PrintWriter pwtr=res.getWriter();
11.        pwtr.println("<B>The selected
subject is: ");
12.        pwtr.println(subject);
13.    }
14. }
```

<b>L i n e N o. 4:</b>	shows SubjectGetServlet extending to HttpServlet
<b>L i n e N o. 5:</b>	uses doGet () with arguments HttpServletRequest req, HttpServletResponse res.
<b>L i n e N o. 7:</b>	uses getParameter () method with “subject” being passed as argument from html file. The program will receive mathematics or physics or chemistry as argument.
<b>L i n e</b>	sets the content as html and Line Nos. 9, 10, 11 send the response to client

**N  
o.  
8  
:**

**Step 3:** Compile the code:

```
C:\PROGRA~1\APACHE~1\TOMCAT~1.0\webapps\  
Demo3\WEB-INF\classes>javac  
SubjectGetServlet.java
```

**Step 4:** create web.xml file in the directory: As per example, 24.16b

**Step 5:** Start the Tomcat server C:\Program Files\Apache Software Foundation\Tomcat 6.0\bin

**Step 6:** Send request on web browser by double clicking on the html icon

**Servlet OUTPUT: The selected subject is: Mathematics**

### *24.15.5 HTTP Post Requests*

We have seen `doGet()` method by `HttpServlet` which is used to get the client request. In this section, we will study `doPost()` method which is used just like

`doGet ()` when fetching input through Form. Html file `SubjectPost.html` and `HttpServlet` program called `SubjectPostServlet.java`. Post request is identical to Get Request except that Form method will have “post” specified explicitly.

**Example 24.19: Apcode.java A Program to Show Usage Post Method in HttpPost Request**

The program uses Post method to send client request and receive response from the server. The application is a simple coding of text in which capital letter is added 13 to get a new character. For example, character ‘A’ will be coded as  $65 + 13 = 78$ , i.e., character ‘B’. If the resultant character goes beyond ‘Z’, then 26 is subtracted. These values are so chosen that characters wrap around when they overflow beyond ‘Z’. Further, observe that we would also shape the response using html document as shown in `Apcode.java` so

that result is displayed using the same form format. We have included coding only for capital letters. There are three distinct files. One is an html file, Apcode.html, and the second file is Apcode.java. These files are created in the directory: C:\Program

Files\Apache Software  
Foundation\Tomcat  
6.0\webapps\examples\ WEB-  
INF\classes

**Web.xml file is in Directory:**

Files\Apache Software  
Foundation\Tomcat 6.0\ webapps\  
examples\ WEB-INF

**Step 1: Create Apcode.html file in directory**

C:\Program Files\Apache Software  
Foundation\Tomcat  
6.0\webapps\examples\ WEB-  
INF\classes

---

```
1. <html>
2. <body>
3. <head><title>Apcode Coder</title>
   </head>
4. <h1>Apcode Coder</h1>
```

```
5. <p>Text to code:
6. <form name="Form1"
7. method="post"
8.
action="http://localhost:8080/examples/A
pcode">
9. <textarea name="text" rows="8"
cols="55">
10. </textarea>
11. <p><input type=submit value =
"code">
12. </form>
13. </body>
14. </html>
```

<b>Line No. 7:</b>	indicates that we are using post method. The action part of the form specifies that the Servlet resource is at examples/web-inf/classes.
--------------------	------------------------------------------------------------------------------------------------------------------------------------------

**Step 2:** Create Servlet program Apcode.java in the directory: C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\examples\ WEB-INF\classes

## Apcode.java

---

```
1. import java.io.*;
2. import javax.servlet.*;
3. import javax.servlet.http.*;
4. public class Apcode extends
HttpServlet {
5. public void doPost(HttpServletRequest
req, HttpServletResponse res)
6. throws ServletException, IOException
{
7. String text =
req.getParameter("text");
8. String translation = translate(text);
9. res.setContentType("text/html");
10. ServletOutputStream out =
res.getOutputStream();
11. out.println("<html>");
12. out.println("<body>");
13. out.println("<head><title>Apcode
Coder</title></head>");
14. out.println("<h1>Apcode
Coder</h1>");
15. out.println("<p>Text to code:</p>");
16. out.println("<form action=\"Apcode\"
method=\"post\">");
```

```
17. out.println("<textarea name=\"text\"  
ROWS=8 COLS=55>");  
18. out.println(translation);  
19. out.println("</textarea>");  
20. out.println("<p><input  
type=\"submit\" value=\"code\">");  
21. out.println("</form>");  
22. out.println("</body>");  
23. out.println("</html>");  
24. }  
25. public void doGet(HttpServletRequest req,  
    HttpServletResponse res)  
26. throws ServletException, IOException  
    {  
27. doPost(req, res);  
28. }  
29. String translate(String input) {  
30. StringBuffer output = new  
    StringBuffer();  
31. if (input != null) {  
32. for (int i = 0; i < input.length();  
    i++) {  
33. char inChar = input.charAt(i);  
34. if ((inChar >= 'A') & (inChar <=  
    'Z')) {  
35. inChar += 13;  
36. if (inChar > 'Z')  
37. inChar -= 26;  
38. }  
39. output.append(inChar);  
40. }  
41. }
```



```

42. return output.toString();
43. }
44. }

```

<b>Line No. 4:</b>	specifies that Apcode extends to <code>HttpServlet</code> interface.
<b>Line No. 5:</b>	shows that <code>doPost()</code> method is used to get the request of the client and send the response.
<b>Line No. 7:</b>	shows usage of <code>req.getParameter("text");</code> to get the client request.
<b>Line No. 9:</b>	sets the content to text / html type.
<b>Line No. 10:</b>	obtains <code>ServletOutputStream</code> object out for sending response to client.

<b>Line Nos . 11 to 23:</b>	show that we are sending response to client using out object and in the same format as that used by Apcode.html
<b>Line No. 16:</b>	Shows use of escape character for setting up of action part of response <pre>out.println("&lt;form action=\"Apcode\" method=\"post\"&gt;");</pre>
<b>Line No. 24:</b>	uses doGet () method and which in turn calls doPost() method at Line No. 27

**Step 3: Compile the code :** C : \Program Files\Apache Software Foundation\Tomcat 6.0\webapps\examples\WEB-INF\classes >javac SubjectPostServlet.java

**Step 4:** create web.xml file in the directory:  
As per example, at 24.16b

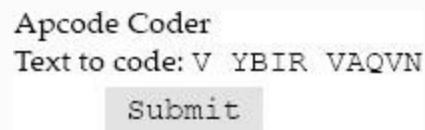
**Step 5:** Start the Tomcat server from  
C:\Program Files\Apache Software  
Foundation\Tomcat 6.0\bin  
directory.

**Step 6:** Send request on web browser by  
double clicking on the html icon

---

OUTPUT form Servlet Apcode:

---



Apcode Coder  
Text to code: V YBIR VAQVN  
Submit

## 24.16 Java Beans

Java's strong point is its reusability feature and java beans is a feature available in Java for providing this feature. Java bean is a portable, platform-independent component model to facilitate writing highly reusable components. For this, we would use Java Beans API. Once these components are developed, we can embed them in applets or applications. Java beans can be visual or non-visual components. Beans are dynamic

in that they can be changed or customized. We can use property of beans component to customize the bean and save the bean using the bean's persistence property.

## **Introspection**

- A Java program can discover the beans property through a process known as introspection.
- A class called Introspector uses core reflection API For introspection to work; we need to follow certain naming conventions and some specific rules so that introspection can identify these properties.
- A bean information class implements the BeanInfo interface. A BeanInfo class explicitly lists those bean features that are to be exposed to application builder tools.
- Properties are the appearance and behaviour characteristics of a bean. Beans expose properties so they can be customized at design time. Properties can be changed at design time by customizers or property editors.
- Bean events can be used for communication between bean components. A listener bean receiving information has to register with source bean.
- Persistence enables beans to save and restore their state.

## **Java Bean API and Important Classes and Interfaces**

- Class Introspector provides interface and method such as `getBeanInfo(Class<type> bean)` throws `IntrospectionException`.

- `PropertyDescriptor` class describes the property of the bean. It supports methods such as `getName()`.
- `EventSetDescriptor` class describes events associated with the bean. It supports methods such as `getAddListenerMethod()` to get names of all addlisteners that bean supports. `getName()` gets you the name of the event.
- `MethodDescriptor` class gives data about methods and supports such as `Method` `getMethod()`.

Normally, several beans are developed by programmers and the third part by developers. In order to reuse them in your applications, you need to know these properties. Hence, these are provided. We can summarize the java beans as:

- Java beans is like a java class.
- Java beans will have properties. In such cases, only the bean class must have `get()` and `set()` methods.
- Java beans must be serializable.
- As beans are part of server side programming, the bean need not be a visual component at client side.

Let us attempt our first bean example. In this example, the instructor welcomes the class. For this, he calls the beans adds, his own message, and fires the beans. It is a traditional welcome students example, used here to highlight the procedures involved. For executing beans programs you will need

a good builder tool like NetBeans 6.1, free downloadable from the net or enterprise version of eclipse that supports beans components. For our example, we have chosen NetBeans 6.1. The concept diagram is shown in Figure 24.6.

**Example 24.20:**  
**Welcomebean.java A Java Beans**  
**Example to Demonstrate the Use of**  
**NetBeans and Beans Components**

We have to use jsp connection because html file is not allowed to call any server components like beans directly. JSP in turn calls for Bean component registered with server like WelcomeBean class.

WelcomeBean class is from a normal java class program.

In an application that does not require any input from html form from the user, you can directly start with jsp module and call class file.

**Step 0:** install NetBean 6.9 version from the net on to your system. Choose the version with full features.

## **Step 1:** Create a new project

---

File→New→javaWeb--→webapplication1→  
next→next→GlashFishServer→ next→  
Finish.

---

**Step 2:** Create Welcome.html file  
WebApplication1→ **webpage** Rightclick→  
new→htmlfile→enter in name field :Welcome  
-→finish

---

```
1. <html>
2. <body>
3. <form method=post action=Welcome.jsp>
4. <input type=text name='text1'>
5. <input type=submit>
6. </form>
7. </body>
8. </html>
```

---

<b>Line No. 3:</b>	html file calls Welcome.jsp.
--------------------	------------------------------

---

Output of html

---

---

### Step 3: Create a jsp file

---

WebApplication1 → webpage Rightclick →  
new → JSP → enter in name field  
:Welcome → finish.

Welcome.jsp

1. <%--
2. Document : Welcome
3. Created on : 14 Jan, 2011, 6:28:39
4. Author : Ramesh
5. --%>
6. <%@page contentType="text/html"  
pageEncoding="UTF-8"%>
7. <!DOCTYPE HTML PUBLIC "-//W3C//DTD  
HTML 01 Transitional//EN"
8. "http://www.worg/TR/html4/loose.dtd">
9. <html>
10. <head>
11. <meta http-equiv="Content-Type"  
content="text/html; charset= UTF-8">
12. <title>JSP Page</title>
13. </head>
14. <body>
15. <jsp:useBean id="Welcomebean"



```

class="jchap24beans.Welcombean" />
16. <%
17. String stg =
request.getParameter("text1");
18. String addlstg =
Welcomebean.welcomeStudents(s);
19. out.println(addlstg);
20. %>
21. </body>
22. </html>

```

<b>Line Nos . 1–14:</b>	are NetBeans generated system document information.
<b>Line No. 15:</b>	Enter user bean id as : "Welcomebean" class = "jchap24beans.Welcombean" />
<b>Line No. 17:</b>	uses <code>getParameter(String msg)</code> format to obtain the request of the client and stores it in String stg.
<b>Line</b>	attaches additional string to the String object

<b>e No. 18:</b>	addlstg by calling a method welcomeStudents(stg) method of Welcomebean class.
<b>Lin e No. 19:</b>	sends out addlstg which is concatenated strings comprising clients request + response i.e. Welcome to Java Beans Students!

#### Step 4: Create Package

WEB-INF → new → java package → jchap24beans  
→ save

#### Step 5: create class

Jchap24beans → new → java class →  
Welcomebean → finish

```

1. package jchap24beans;
2. public class Welcomebean
3. {
4.     public Welcomebean() { }
5.     public String WelcomeStudents(String
s)
6.     {return "Welcome to Java Beans First
Programme..." + s; }
7. }
Welcome to Java Beans.Students!
```

<b>Line No. 4:</b>	Nil argument constructor
<b>Line No. 5:</b>	defines a bean method: WelcomeStudents()

**Step 6:** Save and compile the files in the package.

**Step 7:** Select `Welcome.html` file --> rightclick->run the file **Output on the browser screen: Welcome to Java Beans Students!**

## 24.17 Summary

1. J2SE 5.0 has provided a collection framework. Collection framework is implemented in `java.util` package.
2. Collection class provides several interfaces such as Collection, Map, and Iterator.
3. The Collection Interface is the main interface from which the List and Set Interfaces are derived. It specifies the methods that are common to all the Collections.
4. Set is defined in `java.util.Set`. Set cannot contain duplicate elements.

5. Iterator allows us to iterate over a collection such as Set.
6. List can contain duplicate elements. It supports `ArrayList` and `LinkedList`.
7. The algorithms are implemented as static methods in the `Collections` class.
8. Each entry in the Map involves a pair of elements called keys and values. They are collectively referred to as key-value pairs.
9. Map cannot contain duplicate keys, but can contain duplicate values.
10. JDBC Manager acts as interface between Java programs and database.
11. ODBC (Open Database Connectivity) is provided by Microsoft Corporation. ODBC is shipped along with windows OS. JDBC is a part of JDK which we have installed. It takes calls supplied by JDBC and converts it into formats suitable for proprietary databases such as MS Access and Oracle 10g.
12. MySQL also provides a connector `mysql-connector-java-5.1.3` and `mysql-5.1.51-win32.msi`.
13. DSN stands for Data Source Navigator. The programmers need to create a DSN name to facilitate access to database using ODBC-JDBC Bridge.
14. Oracle database also uses ODBC-JDBC bridge and hence DSN needs to be created.
15. `PreparedStatement` is a pre-parsed and compiled statement that need not be compiled every time statement gets executed. Thus, it is beneficial to use `PreparedStatement` if multiple executions are expected.
16. Callable statements are statements in a java program that call stored procedures. Stored procedures are a set of statements executed on database and results are sent to calling Java program.

17. Servlets like applets are small server side programs that reside on the web servers and extend the web servers functionality across the Internet.
18. Servlet API contains `javax.servlet` and `javax.servlet.http`. These are part of Tomcat `servlet-api.jar`.
19. `ServletRequest` and `ServletResponse` interfaces shown above receive the request object from client and prepares response to be sent.
20. Servlet packages define two types of abstract classes: One is `GenericServlet` belonging to `javax.servlet` and the other is `HttpServlet` belonging to package `javax.servlet.http`.
21. Java beans is a feature available in java for providing this feature. Java bean is a portable, platform-independent component model to facilitate writing highly reusable components.
22. A Java program can discover the beans property through a process known as introspection.
23. A bean information class implements the `BeanInfo` interface. A `BeanInfo` class explicitly lists those bean features that are to be exposed to application builder tools.
24. Properties are the appearance and behaviour characteristics of a bean.
25. Bean events can be used for communication between bean components.
26. Persistence enables beans to save and restore their state.
27. Java beans must be serializable.

## Exercise Questions

---

## Objective Questions

1. Which of the following statements are true in respect of JDBC Connectivity?

1. JDBC-ODBC bridge is required to connect java program to third party databases like MS Access, Oracle, etc.
2. JDBC-ODBC bridge works for MySql database.
3. JDBC-ODBC Bridge is a type 2 Driver
4. Connectivity classes and interfaces are provided by JDBC API

1. i and ii
2. i and iii
3. i and iv
4. iii and iv

2. Which of the following statements are true in respect of JDBC Driver types?

1. Type 1 are vendor specific
2. JDBC-ODBC is a type 2 and vendor independent
3. mysql connector is a type 2 and vendor specific
4. type 4 is a pure java driver and connects directly to database

1. i and ii
2. i and iii
3. i and iv
4. iii and iv

3. Which of the following statements are true in respect of JDBC Driver types?

1. java.sql package contains interfaces and classes for databases
2. Connection Object manages connection between java and database
3. Connection object is used to submit the query
4. Object of statement is used to obtain the result

1. i and ii
2. i and iii
3. i and iv
4. iii and iv

4. Which of the following statements are true in respect of JDBC statements?

1. If repetitive and multiple execution is involved, statement object is inefficient.
2. Prepared statement obviates the need to compile every time in a loop.
3. Callable statement calls procedures and not functions.
4. output by callable statement needs to be registered in order to use it

1. i and ii
2. i, ii and iv
3. i and iv
4. iii and iv

5. Which of the following methods is used by `HttpResponse` interface to dispatch the response to client.

1. `doPost()`
2. `getWriter()`
3. `doGet()`
4. `getResponse()`

6. The action parameter in Form attribute specifies

1. Method at server
2. A procedure at server
3. Servlets at server
4. Applet at server

7. Which of the following statements are true in respect of Java Beans?

1. Platform independent
2. Client side components
3. Server side components
4. Uses JSP (java server pages)

1. i, ii and iii
2. i, ii and iv
3. ii, iii and iv
4. i, iii and iv

8. Which of the following statements are true in respect of Collection Framework of Java?

1. Collection framework has been included in `java.lang` package
2. Collection framework has been included in `java.util` package

3. Iterator is an interface provided by Collections class
4. Collection classes are derived from Collection Object which is derived from Collection class

1. i and ii
2. i and iii
3. i and iv
4. ii, iii and iv

9. Set interface can contain duplicates    TRUE/FALSE

#### **Short-answer Questions**

10. What is ODBC–JDBC Bridge Driver?
11. Distinguish JDBC net driver and pure java driver.
12. Why are type 3 connectors called slow connectors?
13. What is the role of DSN for JDBC-ODBC Connector?
14. Give Java statement for loading and registering Driver by using Class.forName().
15. Give Java statement for establishing the connection.
16. Give an example statement to execute a query.
17. Give Java program statement to insert a record into database.
18. Give java program statement to update a record into database.
19. Give syntax for PreparedStatement.
20. Give syntax of callable statement.
21. Distinguish the Statement and Prepared Statement.
22. Distinguish a Servlet and an Applet.
23. What are the major interfaces of servlet package?
24. What are added to web.xml regarding a servlet?
25. What are the interfaces provided by javax.servlet interface?
26. If java bean file is required to be serializable then what conditions are to be met?
27. What are the features of java beans?
28. What are java beans?
29. What are the features of Set interface?
30. What are Iterators?



31. What are the methods supported by List interface?
32. Explain the Map interface.

#### **Long-answer Questions**

33. Explain the JDBC framework. Explain the usage of various types of database drivers. With explanation, which type is best suited for which type of database?
34. Explain the procedure for creating a DSN for any database like MS Access or Oracles.
35. Explain with Java statements the procedures for loading the Driver, establishing the connection, getting the result through `ResultSet`.
36. Explain the java statements for accessing MySQL databases.
37. Explain the JDBC statements required to insert, update, and delete records from Oracle database.
38. Explain the procedure to use Prepared statement.
39. Explain with suitable examples for using callable statements.
40. Explain `HttpServletRequest` and `HttpServletResponse` interfaces provided by `javax.servlet.http` package.
41. Explain the procedure for obtaining database connection in respect of servlet.
42. Explain the differences and similarities between java class file and beans file.
43. What are the requirements for java bean file?
44. Give out the procedure for executing a java bean.
45. Explain the class and interface and methods of Collection framework of java.
46. What are the important methods supported by Collection Interface?
47. Explain Algorithms interface provided by Collection framework.

#### **Assignment Questions**

48. Write a JDBC connectivity program to connect to Students database at server. The client HTML interface obtains the rollNo from the user and approaches the server. Server to formulate the result marks sheet comprising name Roll No, college name and college code, student name and subject marks.
49. Write a java connectivity program, in which a teacher updates the attendance particulars of the class. A table at Server called attendance has the attendance detail against roll No of the student such as name and dates on which attendance is being taken. The teacher to be able to record attendance either by noting only absentees or by not noting those who are present. The output to be as per format:
- 

Class	Subject Name
subject Code	
Teacher Code	Teacher Name
Roll No	Student Name
date1	
date2	date3.....date60

---

50. Modify the problem at 1 for handling the problem with the use of HttpServlet Request and Response.
51. Write a program with JDBC Connectivity using Servlet to obtain the marks in two sessional examinations and attendance percentage as on particular date. The program to accept roll no from the client and send back response in html document.
52. Write a program with servlet connectivity to display phone book in which client enters the idNo and gets name, number, class, branch , and cell number, email number of both students and parents.

53. Write a servlet-based program to facilitate student registration for a semester. On entering the roll number, program to validate the eligibility and payment of fees and then offer subjects both mandatory and optional and complete the formalities.
54. Write a java beans application in which the user can enter the data about rollNo, name, semester, and server to return registration detail. Use java application.
55. Write a Java program to show the usage of search algorithms provided by Collection framework.
56. Write a program to accept the url from the user. While sourcing the urls, look for other link web pages listed in the url. Store these in the stack and visit each url in turn. Store all the urls visited in a Hash File in the local disk. On demand from the user, perform hash search and provide the url from local HashTable.

### **Solutions to Objective Questions**

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

# Appendix A

## ASCII Table

		0	1	2	3	4	5	6	7	8	9
0	Dec	0	1	2	3	4	5	6	7	8	9
	Hex	0	1	2	3	4	5	6	7	8	9
	Oct	000	001	002	003	004	005	006	007	010	011
	Desc	Null	soh	stx	etx	eot	enq	ack	bel	bl	ht
1	Dec	10	11	12	13	14	15	16	17	18	19
	Hex	A	B	C	D	E	F	10	11	12	13
	Oct	012	013	014	015	016	017	020	021	022	023
	Desc	nl	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	Dec	20	21	22	23	24	25	26	27	28	29
	Hex	14	15	16	017	018	019	1A	1B	1C	1D
	Oct	024	025	026	027	030	031	032	033	034	035
	Desc	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	Dec	30	31	32	33	34	35	36	37	38	39
	Hex	1E	1F	20	21	22	23	24	25	26	27
	Oct	036	037	040	041	042	043	044	045	046	047
	Desc	rs	us	sp	!	"	#	\$	%	&	'
4	Dec	40	41	42	44	44	45	46	47	48	49
	Hex	28	29	2A	2B	2C	2D	2E	2F	30	31
	Oct	050	051	052	053	054	055	056	057	060	061
	Desc	(	)	*	=	/	-	.	/	0	1
5	Dec	50	51	52	54	54	55	56	57	58	59
	Hex	32	33	34	35	36	37	38	39	3A	3B
	Oct	062	063	064	065	066	067	070	071	072	073
	Desc	2	3	4	5	6	7	8	9	:	;
6	Dec	60	61	62	63	64	65	66	67	68	69
	Hex	3C	3D	3E	3F	40	41	42	43	44	45
	Oct	074	075	076	077	100	101	102	103	104	105
	Desc	<	=	>	?	@	A	E	C	D	E
7	Dec	70	71	72	73	74	75	76	77	78	79
	Hex	46	47	48	49	4A	4B	4C	4D	4E	4F
	Oct	106	107	110	111	112	113	114	115	116	117
	Desc	F	G	H	I	J	K	L	M	N	O
8	Dec	80	81	82	83	84	85	86	87	88	89
	Hex	50	51	52	53	54	55	56	57	58	59
	Oct	120	121	122	123	124	125	126	127	130	131
	Desc	P	Q	R	S	T	U	V	W	X	Y
9	Dec	90	91	92	93	94	95	96	97	98	99
	Hex	5A	5B	5C	5D	5E	5F	60	61	62	63
	Oct	132	133	134	135	136	137	140	141	142	143
	Desc	Z	[	\	]	^	_	`	a	b	c
10	Dec	100	101	102	103	104	105	106	107	108	109
	Hex	64	65	66	67	68	69	6A	6B	6C	6D
	Oct	144	145	146	147	150	151	152	153	154	155
	Desc	d	e	f	g	h	i	j	k	l	m
11	Dec	110	111	112	113	114	115	116	117	118	119
	Hex	6E	6F	70	71	72	73	74	75	76	77
	Oct	156	157	160	161	162	163	164	165	166	167
	Desc	n	o	p	q	r	s	t	u	v	w
12	Dec	120	121	122	123	124	125	126	127		
	Hex	78	79	7A	7B	7C	7D	7E	7F		
	Oct	170	171	172	173	174	175	176	177		
	Desc	x	y	z	{		}	~	del		

First read the row, say row No. 6 and then the column, say column No. 5, then the decimal number is 65. From the table intersection, 6 and 5 shows Hex = 41, Oct 101, and character = A.

# Appendix B

## Number Systems

### Introduction

**Decimal System:** We use the decimal system in our daily lives. The allowable digits in decimal system are:

0 1 2 3 4 5 6 7 8 9. The base is 10. The lowest number = 0 ; the highest number = 9. The counting goes 0...99, 100 and so on.

**Binary System:** The computer uses the binary system for its internal representation. The digits allowed are: 0 1. The base is 2. The lowest number = 0 ; the highest number = 1. The counting goes: 0 1 10 11 100 101 110 111 1000... The count is shown in Table B.1.

**Table B.1** Decimal numbers and their binary equivalents

	0	1	2	3	4	5	6	7	8	9
0	0	1	10	11	100	101	110	111	1000	1001
1	1010	1011	1100	1101	1110	1111	10000	10001	10010	10011
2	10100	10101	10110	10111	11000	11001	11010	11011	11100	11101
3	11110	11111	100000							

Computer uses two other representations, namely Octal and Hexa decimal system for addressing mechanisms, storage and display the results to users if required.

**Octal System:** Base is 8. This is equivalent to  $2^3$ . Therefore, 3 digits are required for octal representation. The lowest number = 0; the highest number = 7. The counting is shown below in Table B.2.

**Table B.2** Octal numbers and their binary equivalents

Dec	0	1	2	3	4	5	6	7	8	9
Oct	0	1	2	3	4	5	6	7	10	11
Binary Equivalent	000	001	010	011	100	101	110	111	1010	1011

**Hexa decimal System:** Base is 16. This is equivalent to  $2^4$ . Therefore, 4 digits are required for Hexa-decimal representation. The lowest number = 0; the highest number



= 15. The counting is shown below in Table B.3.

**Table B.3** Hexa-decimal numbers and their binary equivalents  
Appendix C

	Dec	0	1	2	3	4	5	6	7	8	9
0	Hexa Binary Equivalent	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111	8 1000	9 1001
1	Dec Hexa Binary Equivalent	10 A 1010	11 B 1011	12 C 1100	13 D 1101	14 E 1110	15 F 1111	16 10 10000	17 11 10001	18 12 10010	19 13 10011
2	Dec Hexa Binary Equivalent	20 14 10100	21 15 10101	22 16 10110	23 17 10111	24 18 11000	25 19 11001	26 1A 11010	27 1B 11011	28 1C 11100	29 1D 11101
3	Dec Hexa Binary Equivalent	30 1E 11110	31 1F 11111	32 20 100000	33 20 100001						

# Appendix C

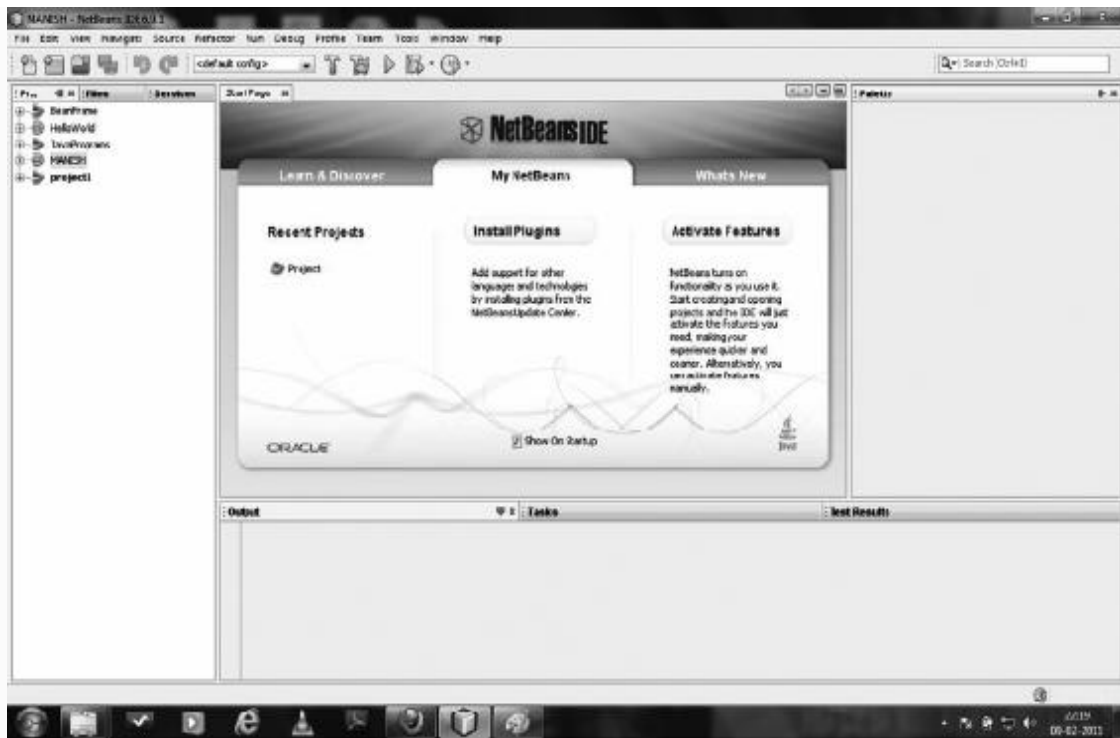
## NetBeans IDE

### Installation

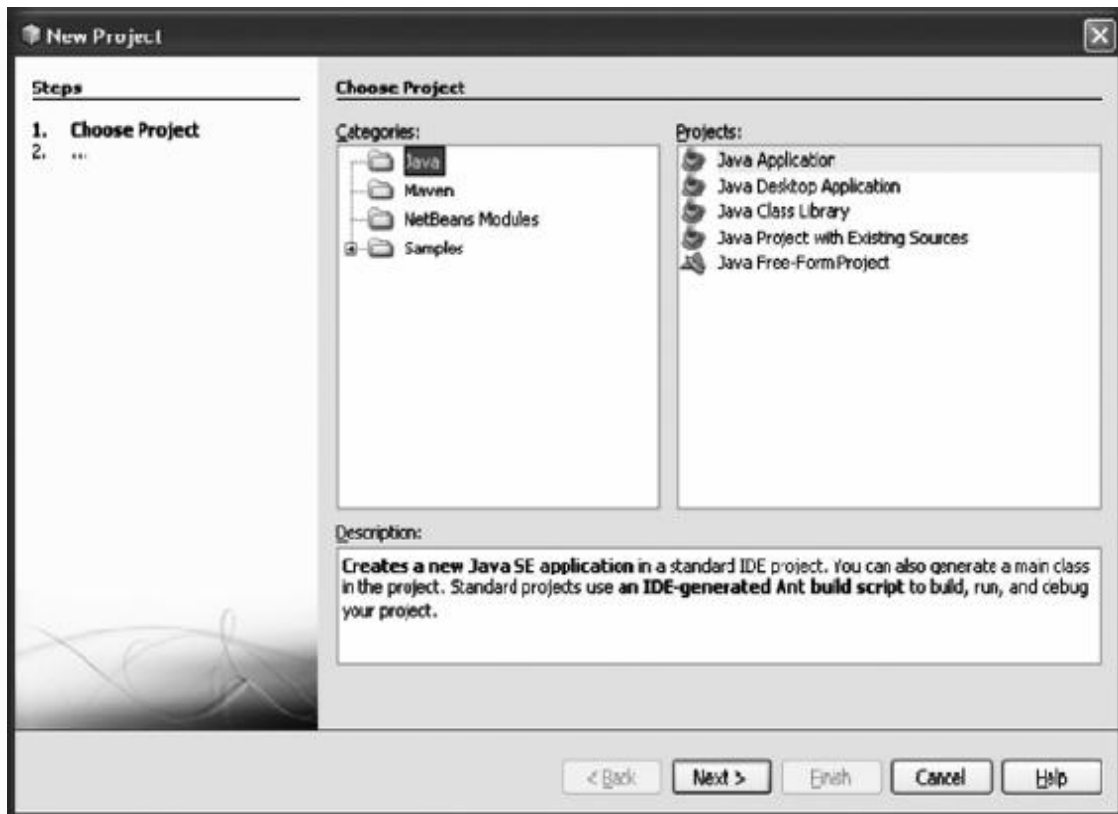
Netbeans 6.9 is an open source and is available at several locations on the Internet. One such location is [www.netbeans.org](http://www.netbeans.org). Down-load and install it on your computer.



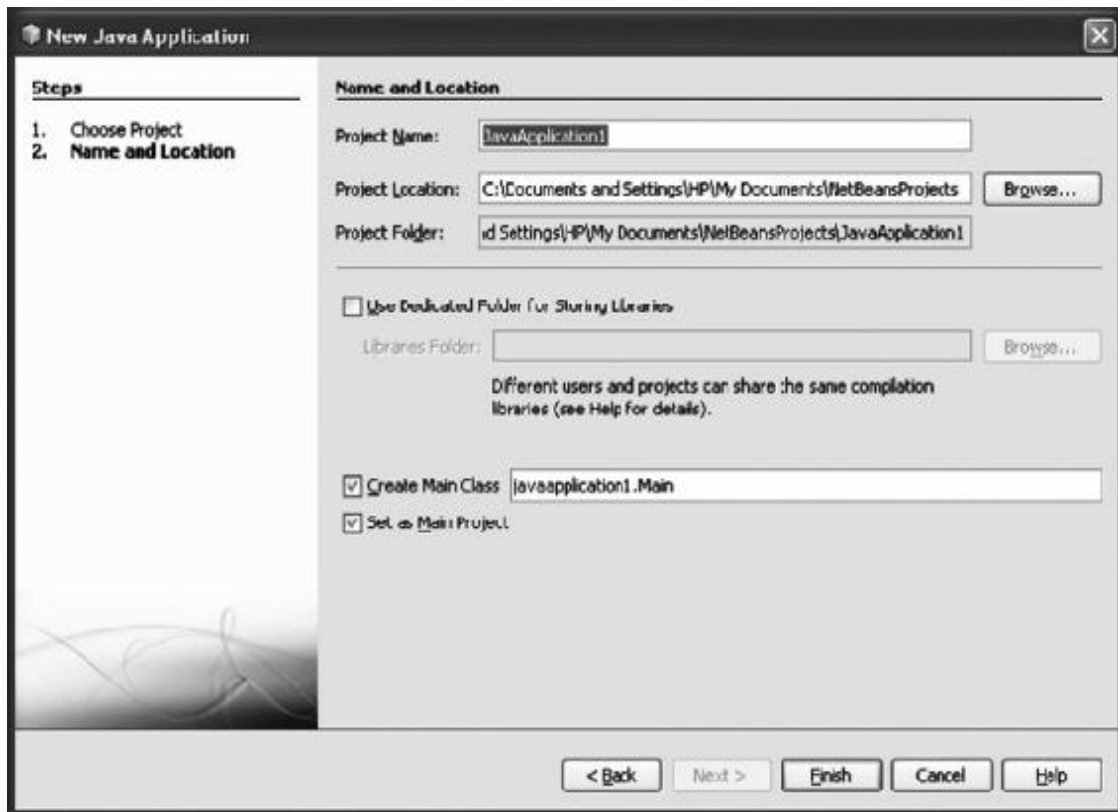
**Step 1:** The first screen that opens on execution of Netbeans package is as shown below:



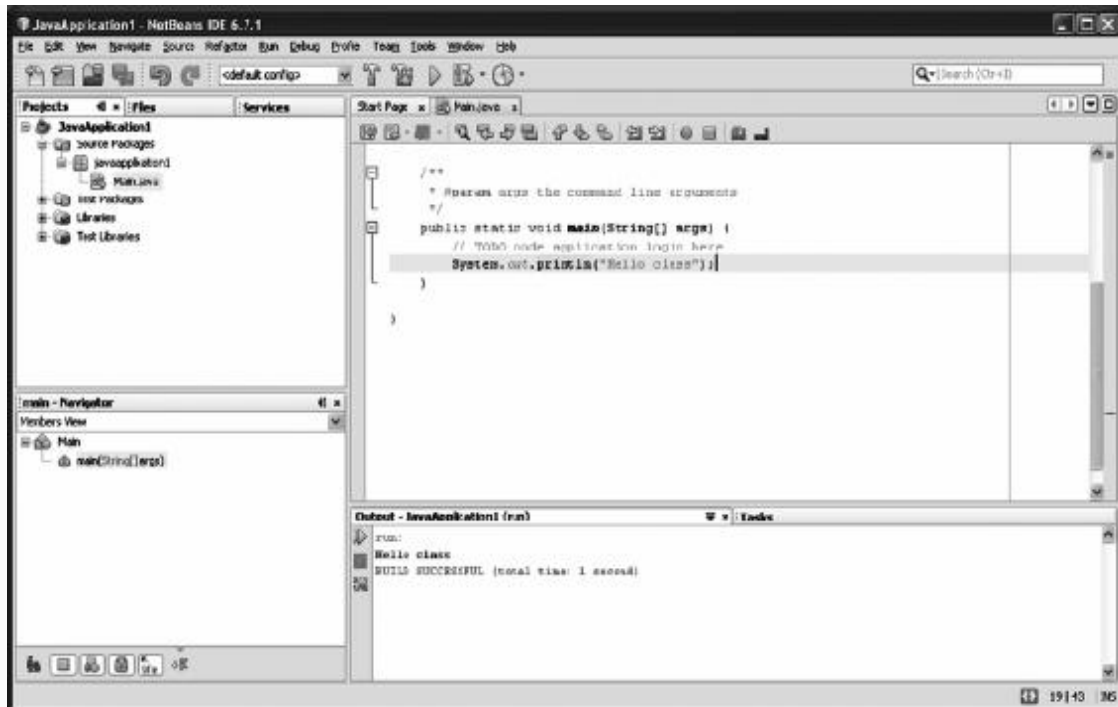
**Step 2:** File → new → java → java Application



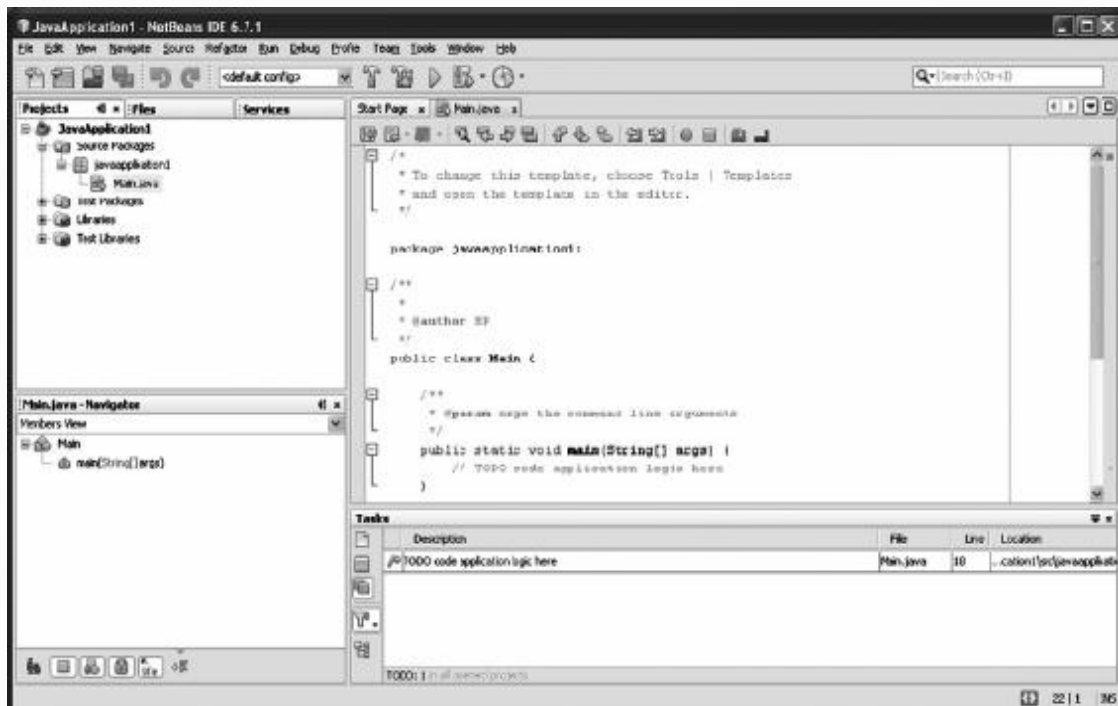
**Step 3:** Select Project Name, Location and folder and select Create Main Class and Set Main Project.



**Step 4:** Enter the java code in the program template provided by Netbeans:



**Step 5:** Select run command from the task bar:







## Acknowledgements

We highly appreciate and express our thanks to our colleagues at Shree Ganpati Institute of Technology for assisting us in testing our solutions. Chiefly, we would like to thank our colleagues Mr Manish Verma, Mr Manish Kumar, Ms Swapna, Mr K. Vishal and Ms Deepika. We are highly indebted to teachers, both from the past and the present, for their valuable suggestions and their painstaking efforts to make this subject easier for students.

We are thankful to Dr M. N. Seetaramnath, formerly, Professor, Computer Science Department, IIT Kharagpur and Andhra University who has taught computer science to us and has been a source of encouragement throughout.

Our special thanks to the Management of Shree Ganpati Institute of Technology

headed by Shri. Susheel Kasana and Shri. S. N. Aggarwal for their encouragement during this project.

We thank the editorial team at Pearson Education—Sojan Jose, S. Shankari, M. E. Sethurajan and Jennifer Sargunar. We highly appreciate the pains they have taken to bring out this high-quality book.

We express our thanks to V. Usha Ramesh for being a tower of support to us. We also thank V. Aparna for her contributions. Finally, we thank Prahlad Jr. for de-stressing us with his playful antics.

**Ramesh Vasappanavara  
Anand Vasappanavara  
Gautam Vasappanavara**

**Copyright © 2011 Dorling Kindersley (India) Pvt. Ltd.**  
Licensees of Pearson Education in South Asia

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

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

ISBN 9788131754559

ePub ISBN 9789332512030

Head Office: A-8(A), Sector 62, Knowledge Boulevard, 7th Floor, NOIDA 201 309, India

Registered Office: 11 Local Shopping Centre, Panchsheel Park, New Delhi 110 017, India