BCBM401T

PROGRAMMING WITH JAVA

DR. C.V. RAMAN UNIVERSITY
Kargi Road, Kota, Bilaspur, (C. G.),
Ph. 07753-253801, 07753-253872
E-mail: info@cvru.ac.in, Website: www.cvru.ac.in

BCBM401T

PROGRAMMING WITH JAVA

Credit- 4

Subject Expert Team

Dr. Ashish Pandey

Assistant Professor, Dr. CV Raman University, Bilaspur

Dr. Rajendra Gupta

Professor, Rabindranath Tagore University, Bhopal

Dr. Ayush Kumar Agrawal

Assistant Professor, Dr. CV Raman University, Bilaspur

Dr. Swarangini Sinha Sinha

Professor, Rabindranath Tagore University, Bhopal

Course Edited By:

Mr. Prateek Saluja
CEO, ResilienceSoft, Bilaspur ,(C.G.)

Unit Written By:

Dr. Vivek Shukla
Associate Professor, Dr. CV Raman University, Bilaspur.

Warning: All rights reserved, No part of this publication may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the publisher.

Edition: 2026

Published by : C.V. Raman University

Kargi Road, Kota, Bilaspur, (C.G.)
Ph. 07753-253802, 07753-253872.

STRUCTURE OF PROGRAM

Title of the Program	**:**	**Bachelor of Computer Application**
Program Type	**:**	**UNDERGRADUATE**
Program Credits	**:**	**120**
Program Duration	**:**	**3 years**
Eligibility Criteria	**:**	**12th Pass**

PROGRAMME OBJECTIVE

- To produce employable IT workforce, that will have sound knowledge of IT and business fundamentals that can be applied to develop and customize solutions for Small and Medium Enterprises (SME).
- To develop skilled manpower in the various areas of information technology like. Data base management, Software Development, Computer-Languages, Software engineering, Web based applications etc.
- To Work professionally with positive attitude as an individual or in multidisciplinary teams and communicate effectively.

PROGRAMME OUTCOME

- **PO 1.** Develop competence in basic technical subjects in computer applications like
  - Programming Languages, Data Structures, Databases, Operating Systems, Software Engineering.
- **PO 2.** Identify, analyze, formulate and develop computer applications.
- **PO 3.** Map real life scenarios to various theoretical optimal solutions.
- **PO 4.** Provide simplest automated solutions to various legacy systems.
- **PO 5.** Use modern computing tools and techniques with confidence.

PROGRAMME SPECIFIC OUTCOME

Upon completion of program in Bachelor of Computer application, students will be able:

- **PSO 1.** Able to implement computer skill in the areas related to multimedia and website design, Data base, Hardware and networking.
- **PSO 2.** Able to demonstrate basic knowledge in the areas such as, Software Engineering, Data communication and Networking, Data base management, Web Technology and Operating Systems for building IT applications.
- **PSO 3.** Students will be able to understand, analyze and develop computer programs in the areas related to algorithm, system software, web design and networking for efficient design of computer-based system.
- **PSO 4.** Apply standard software engineering practices and strategies in software project development using open-source programming environment to deliver a quality of product for business success.
- **PSO 5.** Student will be able to know various issues, latest trends in technology development and thereby innovate new ideas and solutions to existing problems.

STRUCTURE OF COURSE

Title of the Program	**:**	**PROGRAMMING WITH JAVA**
Program Type	**:**	**CORE COURSE**
Course Code	**:**	**BCBM401T**
Course Credits	**:**	**4**
Semester	**:**	**IVth Sem**

COURSE LEARNING OBJECTIVES:

- To introduce and understand students to programming concepts and techniques using the Java language and programming environment, class, objects, also learn about lifetime, scope and the initialization mechanism of variables and improve the ability general problem solving abilities in programming.
- Be able to use the Java SDK environment to create, debug and run simple Java program.

COURSE LEARNING OUTCOMES:

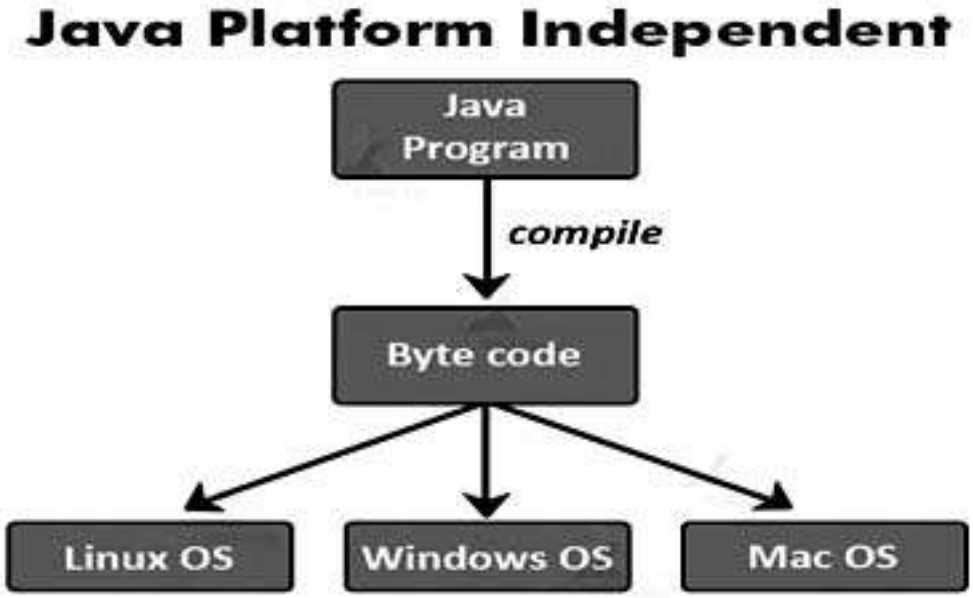
Students will complete software projects comprised of an object-oriented design, implementation, and test plan.

- Designs will demonstrate the use of good object-oriented design principles including encapsulation and information hiding.
- The implementation will demonstrate the use of a variety of basic control structures including selection and repetition; classes and objects in a tiered architecture (user interface, controller, and application logic layers); primitive and reference data types including composition; basic AWT components; file-based I/O; and one-dimensional arrays.
- Test plans will include test cases demonstrating both black box and glass box testing strategies.

COURSE CONTENT

Block I	**FOUNDATIONS OF JAVA PROGRAMMING**	**01 - 50**
Unit 1	**Introduction to Java**	**03**
Unit 2	**Features and Java Environment**	**25**
Unit 3	**Java Language Basics**	**35**
Block II	**JAVA EXECUTION MODEL AND DATA HANDLING**	**53 - 92**
Unit 4	**JVM and Tokens**	**55**
Unit 5	**Variables and Data Types**	**67**
Unit 6	**Type Conversion and Input**	**81**
Block III	**OPERATORS AND DECISION CONTROL**	**95 - 128**
Unit 7	**Operators – I**	**97**
Unit 8	**Operators – II**	**107**
Unit 9	**Control Statements – Decision Making**	**117**
Block IV	**LOOPING STRUCTURES AND CORE DATA STRUCTURES**	**131 – 176**
Unit 10	**Control Statements – Looping**	**133**
Unit 11	**Arrays**	**151**
Unit 12	**String Handling**	**165**
Block V	**OBJECT-ORIENTED BUILDING BLOCKS**	**179 – 222**
Unit 13	**StringBuffer and Wrapper Classes**	**181**
Unit 14	**Classes and Objects**	**193**
Unit 15	**Methods and Constructors**	**209**
Block VI	**ADVANCED OBJECT-ORIENTED AND MULTITASKING CONCEPTS**	**225 – 292**
Unit 16	**Advanced Class Concepts**	**227**
Unit 17	**Inheritance**	**239**
Unit 18	**Polymorphism and Abstraction**	**251**
Unit 19	**Interfaces, Packages and Exceptions**	**253**
Unit 20	**Multithreading, Applets, I/O and Graphics**	**277**

Block I	FOUNDATIONS OF JAVA PROGRAMMING	01 – 50
Unit 1	Introduction to Java	03
Unit 2	Features and Java Environment	25
Unit 3	Java Language Basics	35



Block I introduce learners to the core foundations of Java programming, serving as the entry point for understanding how Java emerged, why it is widely used, and how basic Java programs are written and executed. This block covers Unit 1 (Introduction to Java), Unit 2 (Features and Java Environment), and Unit 3 (Java Language Basics), and is designed to build conceptual clarity before students move to technical depth in later blocks. The block begins by explaining the origin and evolution of Java, highlighting its object-oriented nature, platform independence, and strong security features. Students learn why Java is preferred for developing desktop, web, mobile, and enterprise-level applications. Emphasis is placed on Java's "Write Once, Run Anywhere (WORA)" philosophy, which enables programs to run on different platforms without modification. Next, learners are introduced to the Java programming environment, including components such as the Java Development Kit (JDK), Java Runtime Environment (JRE), and basic tools required to write, compile, and execute Java programs. This helps students understand how Java programs are processed internally and prepares them for hands-on coding.

UNIT – 1
INTRODUCTION TO JAVA

UNIT STRUCTURE

1.1 Introduction
1.2 Objective
1.3 Programming Paradigms and OOPS Concepts
1.4 Evolution and Applications of Java
1.5 Summary
1.6 Glossary
1.7 Answer to check your progress
1.8 Suggested readings – OER
1.9 Reference
1.10 Exercise

1.1. INTRODUCTION

Programming plays a central role in the field of computer applications, as it enables students to convert logical ideas into working software solutions. In earlier semesters, BCA students are typically introduced to basic computer concepts, problem-solving techniques, algorithms, and simple programming approaches. These foundational topics help learners develop logical thinking and understand how a computer executes instructions step by step. Building on this prior learning, this unit introduces Java as a modern, powerful, and widely used programming language that supports structured and object-oriented problem solving.

Java is a high-level, object-oriented programming language designed to be simple, secure, and portable. Unlike low-level languages that depend heavily on hardware, Java allows programmers to focus more on problem logic rather than machine-specific details. One of the most significant features of Java is its platform independence. Java programs are compiled into an intermediate form called bytecode, which is executed by the Java Virtual Machine (JVM). As a result, the same Java program can run on different operating systems without modification, following the principle of "Write Once, Run Anywhere." This feature makes Java especially suitable for internet-based and distributed applications. This unit also introduces the concept of programming paradigms, which are different styles or approaches to designing programs. Students may already be familiar with procedural

NOTES

thinking, where programs are written as a sequence of steps or functions. Java extends this approach by adopting the Object-Oriented Programming System (OOPS), where software is designed around objects that represent real-world entities. Concepts such as class, object, encapsulation, inheritance, polymorphism, and abstraction help in organizing programs in a more natural and reusable manner. Further, the unit discusses the evolution and applications of Java, highlighting how Java has grown from a simple language into a backbone technology for web applications, enterprise systems, and internet services. By the end of this unit, students will be prepared to move toward writing basic Java programs, applying OOPS concepts, and exploring advanced Java features in subsequent units. Thus, this unit acts as a bridge between basic programming knowledge and professional-level software development using Java.

1.2. OBJECTIVE

After completing this unit, the learner will be able to:

- **Identify** the key features and characteristics of the Java programming language
- **Explain** the concept of programming paradigms and classify different types of paradigms
- **Differentiate** between procedural programming and object-oriented programming approaches
- **Describe** core Object-Oriented Programming concepts such as class, object, inheritance, encapsulation, polymorphism, and abstraction
- **Analyze** the advantages of OOPS in solving real-world programming problems
- **Outline** the major stages in the evolution of Java
- **Examine** the role of Java in internet and World Wide Web applications
- **Apply** basic conceptual knowledge of Java and OOPS as a foundation for writing simple Java programs in subsequent units

1.3. PROGRAMMING PARADIGMS AND OOPS CONCEPTS

Programming languages evolve to adapt to changing environments and to implement improvements in the art of programming. For students of Computer Applications (BCA), understanding the shift in these conceptual models is fundamental to mastering modern software development.

1. **Meaning of Programming Paradigms :** A programming paradigm is a conceptual model or approach used to organize and construct a computer program. All computer programs consist of two primary elements: code (the instructions) and data (the information being manipulated). Paradigms determine whether a

program is organized around its code ("what is happening") or around its data ("who is being affected").

- **Process-Oriented (Procedural) Model :** Characterizes a program as a series of linear steps or actions acting upon data.
- **Object-Oriented Model :** Organizes a program around its data (objects) and well-defined interfaces to that data.

2. **The Need for OOPS :** As computer systems transitioned from simple calculators to complex software, the limitations of older paradigms became apparent.
   - **Managing Complexity :** While procedural languages like C were successful, they struggled as programs grew larger. OOPS was conceived to provide better ways to manage this increasing complexity.
   - **Real-World Modeling :** Procedural programming often fails to model real-world problems accurately because functions are action-oriented rather than corresponding to real-world elements.
   - **Security and Robustness :** OOPS provides mechanisms like encapsulation to protect data from unauthorized access or accidental modification.
   - **Software Reusability :** OOPS allows programmers to use existing classes as building blocks, saving time and increasing reliability through pre-tested components.
3. **Transition from Procedural to Object-Oriented Programmin g:** The shift from Procedure-Oriented Programming (POP) to Object-Oriented Programming (OOP) changed the fundamental "unit" of programming.

Tab. No. 1.3: The Procedure-Oriented to Object-Oriented Programming

Feature	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Focus	Emphasis is on actions (doing tasks).	Emphasis is on data (objects).
Structure	Programs are divided into functions.	Programs are divided into methods.
Data Movement	Data moves openly around the program from function to function.	Data is hidden and can only be accessed via specific methods.
Approach	Employs a top-down approach in design.	Follows a bottom-up approach in design.
Complexity	Becomes "spaghetti code" (tangled jumps) as it scales.	Manages complexity through hierarchical classifications.

NOTES

4. **Core OOPS Concepts :** The Object-Oriented Programming Structure (OOPS) is a method of implementation where programs are organized as a collection of objects, classes, and methods.
   - **Class :** A blueprint or template that describes the behaviors and states of an object.

     Example : A "Car" class defining properties like "color" and "engine type."
   - Object: A real-world entity or an instance of a class that occupies memory.

     Example : A specific red Toyota is an object of the "Car" class.
   - **Encapsulation :** Binding code and data together into a protective wrapper to prevent outside interference.
   - **Inheritance :** The process by which one object acquires the properties of another (parent) object, promoting reusability.

     Example : A "Truck" subclass inheriting general traits from a "Vehicle" superclass.
   - **Polymorphism** : The ability of one interface to be used for a general class of actions, allowing one task to be completed in many ways.

     Example : A "draw()" method behaving differently for a Circle than for a Square.
   - Abstraction: The process of hiding internal implementation details and showing only the essential functionality to the user.
5. **Brief Link with Java :** Java is a pure object-oriented language where "everything is an object". It was designed from a "blank slate" to be cohesive and logically consistent with OOPS principles.
   - **Core Java Structure :** In Java, the unit of programming is the class, from which objects are instantiated.
   - **Security and Portability :** Java uses a Virtual Machine (JVM) to execute optimized bytecode, ensuring that programs are secure and can run on any platform ("Write Once, Run Anywhere").
   - **Memory Management :** Unlike C++, Java provides an automatic garbage collector to de-allocate unused objects, making it a robust environment for OOPS

1.3.1. Programming Paradigms

A programming paradigm is the fundamental style, mental map, or conceptual framework used to organize and construct computer programs. It dictates how a programmer perceives the execution of code and the management of information. Historically, computer language innovation has been driven by the need to adapt to

changing environments and to implement refinements in the art of programming. As programs grew and complexity, shifting between paradigms became necessary to maintain robustness and manageability.

1. **Procedural Programming (The Process-Oriented Model):** The procedural paradigm, often referred to as the process-oriented model, characterizes a program as a series of linear steps. In this model, the primary focus is on actions or "what is happening".

   Characteristics:

   - **Action-Oriented:** The unit of programming is the function or subroutine.
   - **Linear Execution:** Programs are seen as a sequence of tasks to be performed.
   - **Global Data:** Many important data items are placed in a global scope so they can be accessed by all functions, which makes them vulnerable to inadvertent changes.
   - **Top-Down Design:** Employs a top-down approach where a large problem is broken into smaller pieces (stepwise refinement) until the tasks can be solved directly.
   - **Data vs. Code:** The model is conceptually defined as code acting on data.

   Drawbacks : This paradigm often fails to model real-world problems effectively because functions do not always correspond to actual problem elements. As programs grow, they may devolve into "spaghetti code" a tangled mass of jumps and branches that are nearly impossible to understand or maintain.

   Examples : High-level languages such as C, COBOL, FORTRAN, and Pascal are classic examples of procedural programming.

2. **Object-Oriented Programming (The Data-Oriented Model):** To manage the increasing complexity of large-scale software, the object-oriented programming (OOP) paradigm was conceived. It organizes a program around its data (objects) rather than the logic of its actions.

   Characteristics :

   - **Data-Oriented:** The focus is on "who is being affected".
   - **Data Controls Code:** An object-oriented program is characterized as data controlling access to code.
   - **Bottom-Up Design:** It follows a bottom-up approach, building complex systems from smaller, reusable modules.
   - **Real-World Modeling:** OOD (Object-Oriented Design) provides a natural way to view the software process by modeling attributes and behaviors of objects.

NOTES

Core Principles :

- **Encapsulation:** The mechanism that binds code and data together, keeping both safe from outside interference via a "protective wrapper" or class.
- **Inheritance:** The process by which one object acquires the properties of another, supporting hierarchical classification and code reusability.
- **Polymorphism:** A feature allowing one interface to be used for a general class of actions, often summarized as "one interface, multiple methods".
- **Abstraction:** The process of hiding implementation details and showing only essential functionality to the user to manage complexity.

  Examples: Java and C++ are prominent object-oriented languages.

3. **Functional Programming:** Functional programming is a paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. In Java, this was significantly introduced with the addition of lambda expressions.

   Characteristics:

   - **Declarative Style:** Focuses on *what* to solve rather than *how* to solve it through explicit steps.
   - **Pipeline Operations:** Supports pipeline operations on data through streams, allowing for efficient mapping and filtering.
   - **Statelessness:** Often emphasizes the use of effectively final variables to avoid side effects.
   - **Code as Data:** Enables passing executable code as arguments to methods, enhancing expressive power.

     Examples: Modern versions of Java (specifically Java SE 8 and later) utilize functional features through the java.util.stream and java.util.function packages.

4. **Logical Programming:** The following information regarding the Logical Paradigm is not explicitly detailed in the provided sources and should be independently verified. The logical paradigm is based on formal logic. A programmer defines a set of facts and rules (the knowledge base), and the computer uses an inference engine to find solutions to queries.

   Characteristics:

   - **Rule-Based:** Programs consist of logical statements rather than sequences of instructions.
   - **Goal-Oriented:** The user provides a goal, and the system attempts to prove it based on available facts.

     Example: Prolog is the most well-known logical programming language.

Tab. No. 1.3.1: Comparison Table of Paradigms

Feature	Procedural (POP)	Object-Oriented (OOP)
Primary Unit	Function/Subroutine	Object/Class
Emphasis	Actions (Doing tasks)	Data (Modeling entities)
Data Handling	Global/Local Data	Encapsulated (Private)
Design Approach	Top-Down	Bottom-Up
Control Flow	Sequential steps/Loops	Method calls/Messaging
Reusability	Low (Task-specific)	High (Inheritance / Classes)

Simple Programming Examples

Procedural Example (C-Style logic) : In a procedural approach, computing the area of different shapes might involve a large switch or if statement within a single function.

```
// Logic: Code acting on data
void calculateArea(int type, double dim1, double dim2) {
    if (type == 1) // Rectangle
        System.out.println(dim1 * dim2);
    else if (type == 2) // Triangle
        System.out.println(0.5 * dim1 * dim2);
} [Based on 239, 712]
```

Object-Oriented Example (Java): In OOP, each shape is an object that knows how to calculate its own area using polymorphism.

```
abstract class Shape {
    abstract double area();
}
class Rectangle extends Shape {
    double w, h;
    double area() { return w * h; }
} [Based on 858, 1714]
```

Functional Example (Java Lambda) : Using a functional approach, you can pass the logic as an argument or apply it to a stream of data

1.3.2. Object Oriented Programming concepts

Object-Oriented Programming (OOP) is a methodology or paradigm used to design computer programs by organizing them as collections of objects, classes, and methods. Unlike procedural programming, which characterizes a program as a series of linear steps acting on data, OOP organizes a program around its data (objects) and well-defined interfaces to that data.

1. **Class:** A Class is a logical construct that acts as a template or blueprint for creating objects. it defines the structure and behavior (data and code) that will be shared by a set of objects of its type.

NOTES

Example : A class is like a blueprint for a house. you can use one blueprint to build many physical houses.

2. **Object:** An Object is an instance of a class and represents a physical as well as logical entity. It has a state (represented by values), behavior (represented by functionalities), and identity (a unique internal ID used by the JVM).

   Example: If "Person" is a class, then a specific person like "John" is an object with unique attributes like age and weight.

3. **Encapsulation:** Encapsulation is the mechanism that binds together code and the data it manipulates, keeping both safe from outside interference or misuse. It is often described as a protective wrapper that enforces information hiding, where implementation details are hidden within the object.

   Example: An automatic transmission in a car; you use the gear-shift lever (interface) without needing to know the internal mechanics of the engine or transmission.

4. **Abstraction:** Abstraction is the process of hiding implementation details and showing only the essential functionality to the user. It helps manage complexity by allowing the user to utilize an object without being overwhelmed by its internal parts.

   Example: When you send an SMS, you type the text and hit send; you do not need to know the internal processing or network delivery protocols.

5. **Inheritance:** Inheritance is the procedure by which one object acquires the properties and behavior of another. It is used to achieve code reusability and to reduce memory usage. In Java, this is implemented using the extends keyword.

   Example: A Golden Retriever is a specific type of Dog, which is a type of Mammal, which is a type of Animal; it inherits general traits from its ancestors.

Types of Inheritance:

1. **Single:** One parent class to one child class.
2. **Multilevel:** A child class is derived from another child class.
3. **Hierarchical:** Multiple child classes from one parent.
4. **Multiple:** One child class from multiple parents (only supported in Java via **interfaces** to avoid priority problems).
5. **Hybrid:** A combination of single and multiple inheritance.
6. **Polymorphism:** Polymorphism (Greek for "many forms") is the feature that allows one interface to be used for a general class of actions. It means a single task can be completed in many ways.

Example : A dog's sense of smell is polymorphic; it reacts differently depending on whether it smells a cat (chase) or food (salivate).

Types of Polymorphism:

1. **Compile-time (Static Binding):** Achieved through Method Overloading, where methods have the same name but different parameters (number, type, or order) in the same class.
2. **Run-time (Dynamic Binding):** Achieved through Method Overriding, where a subclass provides a specific implementation for a method already defined in its superclass.

1.3.3. Advantages of OOPS

Object-Oriented Programming (OOPS) is a methodology used to design programs using classes and objects, which simplifies software development and maintenance by providing a structured approach to coding.

Advantages Over Procedural Programming: OOPS was conceived specifically to manage the increasing complexity of large software systems, a task where traditional process-oriented models (like C) often fail. The following table highlights the key differences between Procedure Oriented Programming (POP) and Object-Oriented Programming (OOP).

Tab. No. 1.3.3 blank table

Feature	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Focus	Emphasis is on doing actions and functions.	Emphasis is on data and objects.
Structure	Large programs are divided into smaller functions.	Programs are divided into classes and methods.
Data Security	Data moves openly around the system and is vulnerable to accidental change.	Data is hidden and can only be accessed by associated methods.
Approach	Employs a top-down approach in design.	Follows a bottom-up approach in design.
Maintenance	Code maintenance and reuse is difficult.	Code maintenance and reuse is easy.
Upgradability	Not easy to upgrade.	Easy to upgrade.

Maintainability: OOPS provides a natural and intuitive way to view the design process by modeling objects based on their attributes and behaviors.

Modularization: Programs are decomposed into independent entities (objects) that can be used in different programs without modifications.

Information Hiding: By hiding implementation details within objects, a programmer can modify the "inside" of a subroutine or class without affecting the rest of the program, provided the interface remains unchanged.

NOTES

Java Application Examples: Java and OOPS principles are utilized across various domains.

- **Enterprise Applications:** Primarily used in banking systems and large-scale distributed networking applications.
- **Mobile Applications:** Android apps and J2ME-based mobile platforms.
- **Web Applications:** Dynamically generated content using Servlets and JSPs.
- **Embedded Systems:** Found in smart cards, microwave ovens, and remote controls.
- **Gaming and Robotics:** Used extensively for game development and robotic control systems.

1.4. EVOLUTION AND APPLICATIONS OF JAVA

Java is a high-level, object-oriented programming language originally developed by Sun Microsystems and released in 1995. Today, it is owned by Oracle and runs on more than 3 billion devices across the world. Java was conceived in 1991 by a team led by James Gosling. The language, which was originally called "Oak," was created for a specific purpose: to develop software for consumer electronic devices, such as microwave ovens and remote controls. The main problem with other languages like C and C++ at the time was that they were designed to be compiled for specific targets or CPUs. This meant a program had to be re-written or re-compiled for every new device, which was expensive and time-consuming. The Java team wanted to create a platform-independent language that could produce code that would run on any CPU under different environments. As the project progressed, the World Wide Web became popular, and the team realized that the Internet faced the same problems as small electronics: it needed programs that were portable and secure across many different types of computers. This led Java to shift its focus toward Internet programming.

Features that Made Java Popular : Several key features, often called "buzzwords," helped Java become one of the most successful languages in history.

- **Platform Independent :** Java code is not compiled into a machine-specific language; instead, it becomes "bytecode". This bytecode is interpreted by the Java Virtual Machine (JVM), allowing programs to follow the motto: "Write Once, Run Anywhere" (WORA).
- **Object-Oriented:** Java organizes programs around data (objects) rather than just actions, which makes code easier to reuse and maintain.
- **Secure:** Java provides a "virtual firewall" between the application and the computer. It lacks features like pointers, which can be misused to access restricted memory, making it safer for downloading programs over a network.

- **Simple:** It was designed to be easy to learn by using a syntax like C and C++ while removing complex features like manual memory management.
- **Robust:** Java is built to be reliable; it manages memory for you through automatic garbage collection and performs strict checks to catch errors early.
- **Multithreaded:** This feature allows a program to perform several tasks simultaneously, such as printing a document while you are still typing in a text editor.
- **Areas of Application :** Because of its flexibility and security, Java is used in almost every field of technology.
- **Desktop Applications:** Traditional software installed on computers.
- **Web Applications:** Creating dynamic websites and server-side programs using Servlets.
- **Mobile Applications:** Java is widely used for mobile development, with billions of Java-enabled hand-held devices in use.
- **Enterprise Solutions:** Large-scale, distributed systems for big businesses and banking.
- **Embedded Systems:** Software found in appliances, smart cards, and industrial machinery.
- **Scientific and Big Data:** Used in robotics and high-performance applications like Big Data processing.
- **Games:** Many popular games are developed using Java's robust environment.

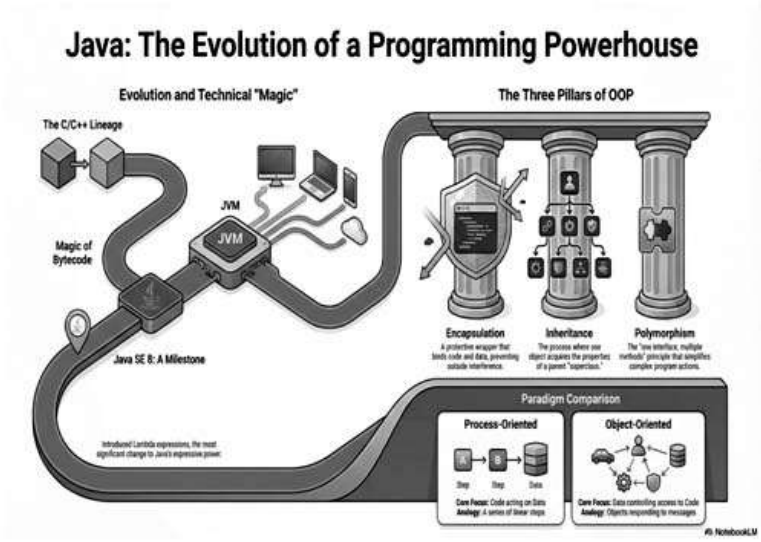


Fig. 1.1 : History of Java.

1.4.1. Evolution of Java

The evolution of Java is a journey that transformed a small-scale project for consumer electronics into the premier programming language for the global Internet and enterprise computing. The development of Java was driven by two fundamental factors: the need

NOTES

to adapt to changing computing environments and the desire to implement refinements in the art of programming. Java was conceived in 1991 at Sun Microsystems, Inc. It was initially designed by a small group of engineers known as the "Green Team," which included James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan. The original impetus for the language was not the Internet; rather, the team sought to create a platform-independent (architecture-neutral) language for use in embedded systems. Specifically, they wanted a common development environment for consumer electronic devices, such as microwave ovens, remote controls, and television set-top boxes. At the time, languages like C and C++ were designed to be compiled for a specific target CPU, making the creation of software for diverse hardware expensive and time-consuming. Gosling and his team sought a more cost-efficient solution where code could run on any variety of CPUs under differing environments. James Gosling and the Naming of Java James Gosling is widely recognized as the lead inventor and initiator of the Java language project. Under his leadership, it took approximately 18 months to develop the first working version.

The naming of the language underwent an interesting evolution :

- **GreenTalk :** Initially, the language was named GreenTalk, and its source files used the .gt extension.
- **Oak :** Gosling later renamed the language "Oak" after an oak tree that stood outside his office at Sun Microsystems. Oak is also the national tree of several countries, including the USA and France, representing solidarity. However, the name was discarded because "Oak Technologies" already held a trademark for that name.
- **Java :** In 1995, the team brainstormed new names, including Silk, DNA, Ruby, and Java. "Java" was eventually selected from a list of random words. Gosling proposed the name based on the Java coffee bean (sourced from the island of Java in Indonesia), as it represented the essence of being dynamic, revolutionary, and fun to say.

Key Milestones in Java Development

- **1991:** Initiation of the Java language project at Sun Microsystems.
- **1992:** Implementation of the initial "Oak" version.
- **1995:** Public announcement and release of Java 1.0 (J2SE). This version promised the revolutionary "Write Once, Run Anywhere" (WORA) capability.
- **2006:** On November 13, Sun released much of Java as free and open-source software under the terms of the GNU General Public License (GPL).
- **2007:** Sun finished making almost all of Java's core code free and open source.

- **2010:** Oracle Corporation acquired Sun Microsystems, taking ownership of the Java platform.

Major Java Versions (Conceptual Overview) : The evolution of Java is marked by significant releases that expanded its capabilities.

- **JDK 1.1 (1997) :** Introduced major features such as JavaBeans, RMI, JDBC, and inner classes.
- **JDK 1.2 / Java 2 (1998):** This was a "watershed event" marking the beginning of Java's "modern age". It introduced Swing for GUI development, the Collections Framework, and the Just-In-Time (JIT) compiler. Sun rebranded the platform as J2SE (Java 2 Platform Standard Edition).
- **J2SE 1.4 (2002):** Enhanced Java with the assert keyword, chained exceptions, and a channel based I/O (NIO) subsystem.
- **J2SE 5 / JDK 5 (2004):** A revolutionary release that fundamentally expanded the language's scope. Key additions included Generics, Annotations, Autoboxing/Unboxing, Enumerations (enums), and the enhanced for-each loop.
- **Java SE 7 / JDK 7 (2011):** The first release under Oracle. It introduced Project Coin features (such as String in switch statements and try-with-resources for automatic resource management) and the Fork/Join Framework for parallel programming.
- **Java SE 8 / JDK 8 (2014):** A profound upgrade that added Lambda Expressions, bringing functional programming features to Java. It also introduced the Stream API for pipeline operations on data and the Optional class.
- **Modern Releases :** Java now follows a six-month release cycle, with recent versions including Java SE 21, 22, and 23, which continue to refine features like pattern matching and generational garbage collection.

Reasons for Java's Success : Java's rapid ascent and enduring popularity can be attributed to several key characteristics, often referred to as "buzzwords":

- **Platform Independence :** Java code is compiled into bytecode, an intermediate representation. This bytecode is executed by the Java Virtual Machine (JVM), which acts as an interpreter or a JIT compiler on the host system. This architecture allows the same program to run on any device with a compatible JVM without recompilation (WORA).
- **Security:** Java was designed for networked environments. It achieves security by confining programs (like applets) to a sandbox execution environment, preventing them from accessing private system resources like the local file system without authorization. It also eliminated dangerous features like explicit pointers, which could be used to breach memory firewalls.

NOTES

- **Simplicity :** Java inherits its syntax from C and C++, making it familiar to experienced programmers. However, it removed complex and error-prone features such as operator overloading, multiple inheritance of classes, and manual memory management.
- **Robustness :** Java emphasizes reliability through strict compile-time and run-time checking. Its automatic garbage collection eliminates memory leaks by deallocating memory for objects that are no longer in use. It also provides a structured approach to error handling through exceptions.
- **Multithreading :** Unlike many languages that rely on the operating system for multitasking, Java has built-in support for multithreading, allowing a single program to perform multiple tasks simultaneously.
- **Open Source & Community Driven :** Sun's decision to make Java open source under the GNU GPL encouraged widespread innovation and transparency. The Java Community Process (JCP) allows for new ideas to be assimilated into the language systematically.
- **Distributed Nature :** Java was designed with the Internet in mind, supporting TCP/IP protocols and providing features like Remote Method Invocation (RMI) for objects to interact across a network.

1.4.2. Java and Internet

The Internet is a global network that enables computers and devices to communicate and share information using standardized protocols such as TCP/IP. Java was specifically designed to work efficiently in this networked environment. From its inception, Java focused on providing a secure, platform-independent, and robust programming language capable of supporting Internet-based applications. The strong relationship between Java and the Internet made Java one of the most widely used languages for web-based and distributed systems. Java supports Internet programming primarily through its ability to create network-aware applications**.** Unlike traditional programming languages that were limited to standalone systems, Java applications can easily communicate across networks. Java provides built-in libraries such as java.net, which allow programmers to work with URLs, sockets, HTTP connections, and data streams. This makes Java ideal for developing applications like online forms, chat systems, email clients, and cloud-based services. One of the key features that connects Java with the Internet is platform independence**.** Java programs are compiled into bytecode, which runs on the Java Virtual Machine (JVM). Since JVMs are available for multiple operating systems, Java programs can run on any Internet-connected device without modification. This feature supports the philosophy of "Write Once, Run Anywhere (WORA)", which is

extremely valuable for Internet applications accessed by users on different platforms. Another important aspect is security**.** Internet applications are exposed to untrusted networks and users. Java includes security mechanisms such as bytecode verification, class loaders, and security managers that prevent unauthorized access to system resources. This ensures safer execution of Internet-based Java programs. Java also supports multithreading, which is essential for Internet applications. Web servers and online applications often handle multiple users simultaneously. Java's built-in multithreading capability allows applications to process multiple client requests efficiently without slowing down system performance. Java Internet programming is used in applications such as:

- Online banking systems
- E-commerce platforms
- Web-based management systems
- Distributed enterprise applications

Thus, Java plays a vital role in Internet-based computing by providing reliability, portability, security, and scalability.

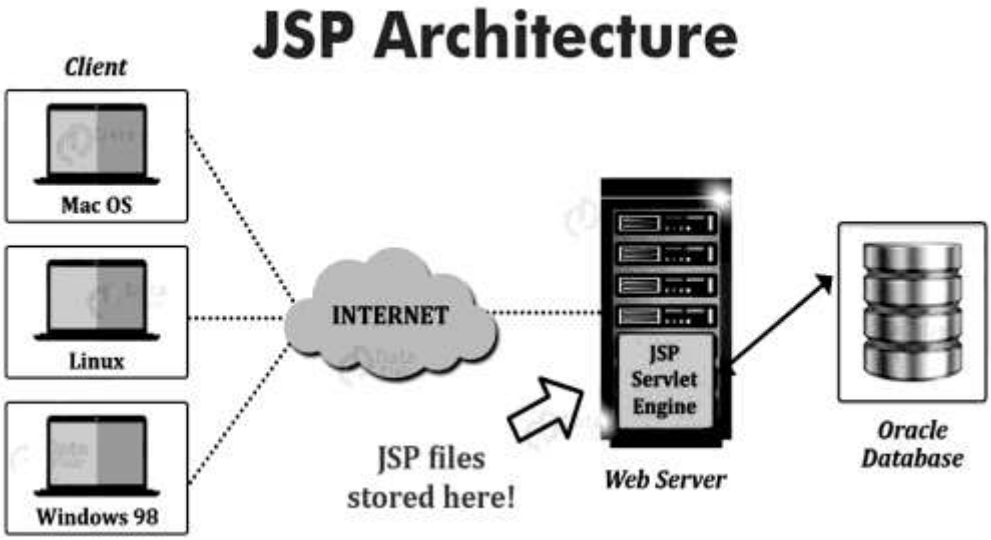


Fig. 1.4.2 (a) : Java in Internet Environment

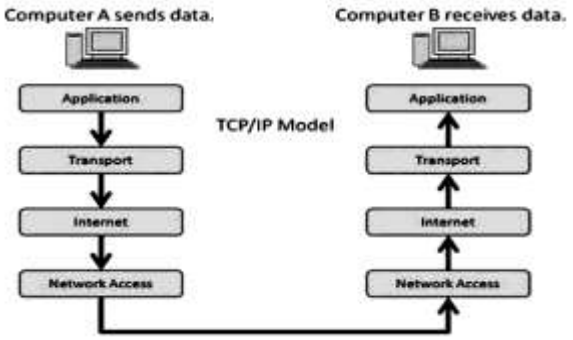


Fig. 1.4.2 (b) : Java in Internet Environment.

NOTES

Types of Java Internet Applications: Java supports various types of Internet-based applications depending on the execution environment and user interaction. These types enable Java to be used both on the client side and server side.

1. **Standalone Internet Applications**: These applications use the Internet only for communication but run independently on a local machine. Examples include email clients and file transfer applications. Java sockets and networking APIs are commonly used for such programs.
2. **Applets (Historical Importance)**: Java applets were small programs embedded in web pages and executed within a browser. Although applets are now obsolete due to security issues, they played a significant role in demonstrating Java's Internet capabilities. Applets allowed dynamic content and interactive web pages.
3. **Server-Side Java Applications**: These applications run on web servers and generate dynamic content. Examples include Java Servlets and Java Server Pages (JSP). They process client requests and send responses using HTTP protocol.
4. **Distributed Applications :** Java supports distributed computing using technologies like RMI (Remote Method Invocation). These applications allow objects running on different machines to communicate over the Internet.

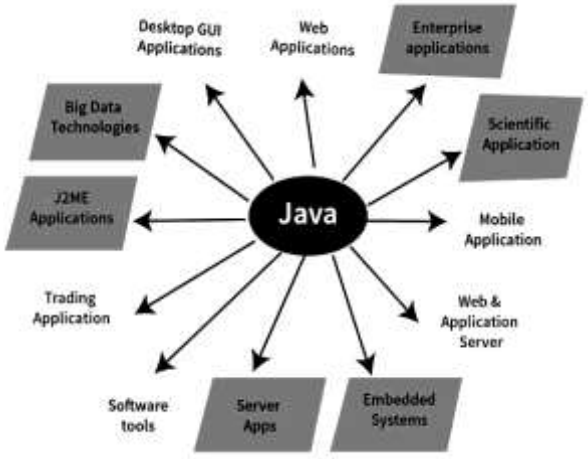


Fig. 1.4.2.1: Types of Java Internet Applications

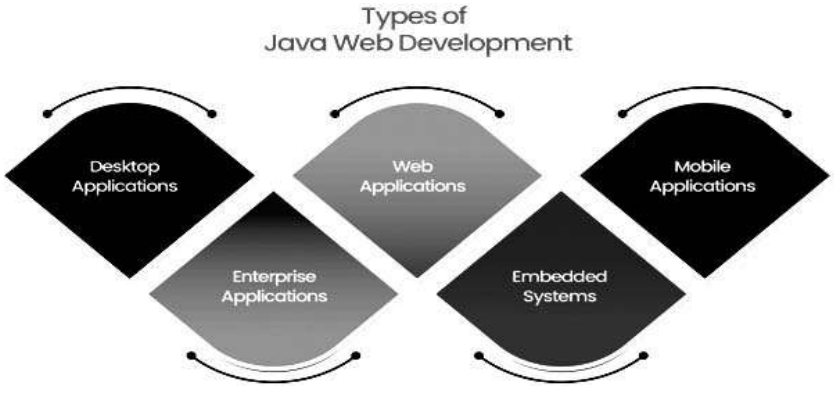


Fig. 1.4.2.2 : Types of Java Internet Applications.

Examples of Java Internet Usage : A simple example of Java Internet programming is accessing a web resource using a URL:

- Java can read data from a website
- Java can send requests to servers
- Java can build real-time chat or messaging systems

These examples highlight Java's strength in Internet programming and its continued relevance in modern web systems.

1.4.3. JAVA AND WORLD WIDE WEB (WWW)

The World Wide Web (WWW) is a system of interlinked web pages accessed through the Internet using browsers. Java has played a significant role in enhancing the functionality, interactivity, and dynamic behavior of web applications. While HTML provides structure and CSS controls presentation, Java enables logic-driven and interactive web systems. Java is mainly used on the server side of the WWW, where it processes user requests, interacts with databases, and generates dynamic web pages. Technologies such as Servlets, JSP, and frameworks like Spring allow Java to build powerful web applications. When a user requests a webpage, the browser sends an HTTP request to the server, where Java programs process the request and return an appropriate response. One of the major strengths of Java in WWW is scalability. Java-based web applications can handle thousands of users simultaneously, making them suitable for enterprise-level websites. Java's multithreading and memory management support efficient request handling. Another important aspect is portability and maintainability. Java-based web applications can be deployed on different servers such as Apache Tomcat, GlassFish, or JBoss without code changes. This flexibility reduces development and maintenance costs. Java also ensures security in web applications. Features like authentication, authorization, encrypted communication (HTTPS), and secure session management are easily implemented using Java technologies. This makes Java suitable for sensitive web applications like online banking, e-governance portals, and university management systems.

In modern web development, Java works in combination with:

- HTML, CSS, JavaScript (Frontend)
- Databases like MySQL or Oracle
- Web servers and application servers

Thus, Java forms the backbone of many web-based systems operating on the World Wide Web.

NOTES

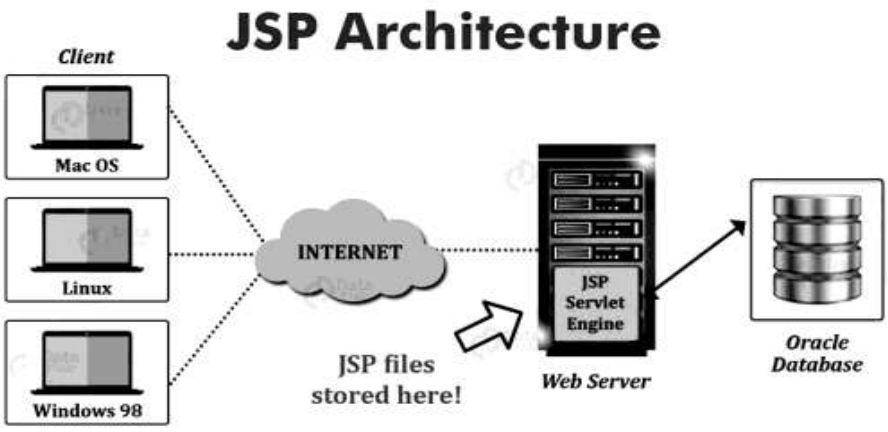


Fig. 1.4.3 (a) : Java-Based Web Application Architecture.

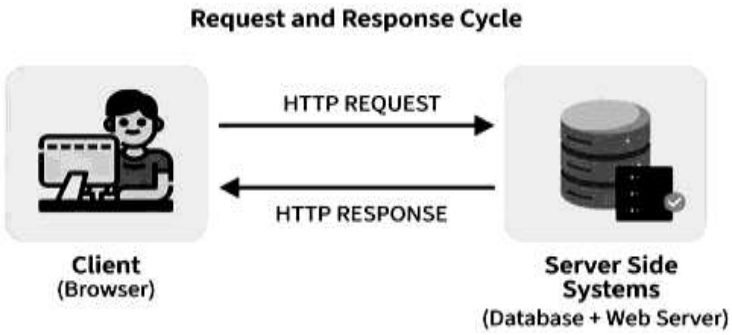


Fig. 1.4.3 (b) : Java-Based Web Application Architecture

Java Technologies Used in WWW

1. **Java Servlets :** Servlets are Java programs that run on web servers and handle client requests. They replace traditional CGI programs and provide better performance.
2. **Java Server Pages (JSP) :** JSP allows embedding Java code into HTML pages to create dynamic web content.
3. **Java Frameworks :** Frameworks like Spring and Hibernate simplify web application development by providing reusable components and structured architecture.

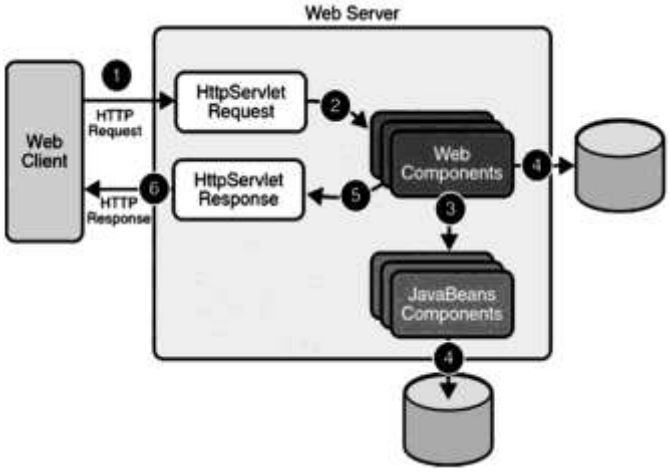


Fig. 1.4.3.1: Java Technologies in WWW.

Examples of Java in WWW Applications:

- Online admission systems
- Learning management systems (LMS)
- E-commerce websites
- Online examination portals

These applications demonstrate how Java enables dynamic content generation, secure transactions, and efficient user interaction on the www.

Activity 1.1 :

Exploring OOP Concepts through Real-Life Examples

Objective :

To help students understand OOP concepts by relating them to real-world scenarios.

Instructions :

- Choose any real-life system (e.g., library system, banking system, or school management system).
- Identify at least three objects from the system.
- Describe their attributes and behaviors.
- Explain how OOP concepts like inheritance, encapsulation, or polymorphism can be applied to your chosen system.
- Present your findings in a short report (150–200 words) or a diagrammatic representation.

☑ Check Your Progress 1.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What is Java?

. . . . . .

. . . . . .

. . . . . .

2. What is a programming paradigm?

. . . . . .

. . . . . .

. . . . . .

3. Name any two OOPS concepts.

. . . . . .

. . . . . .

. . . . . .

NOTES

1.5. SUMMARY

This unit provides a clear and foundational understanding of Java and Object-Oriented Programming concepts for BCA students. It begins with an introduction to Java as a high-level, object-oriented, and platform-independent programming language that follows the principle of "Write Once, Run Anywhere." The objectives of the unit focus on helping students understand the basics of Java, identify different programming paradigms, and learn how Object-Oriented Programming System (OOPS) concepts are used to solve real-world problems efficiently. The unit explains programming paradigms and highlights how object-oriented programming differs from procedural approaches by organizing programs around objects rather than functions. Core OOPS concepts such as class, object, encapsulation, inheritance, polymorphism, and abstraction are discussed in simple terms to help students model real-world entities in software development. The evolution of Java is also covered, tracing its development by James Gosling at Sun Microsystems and its growth into a widely used language for internet and web-based applications. The role of Java in internet programming and the World Wide Web is explained, emphasizing its use in secure, scalable, and dynamic applications. Overall, this unit builds a strong conceptual base in Java and OOPS, preparing students for advanced programming topics and practical application development.

1.6. GLOSSARY

A list of new and difficult terms used in the text is to be given at the end part of the unit. Each word, term or concept is explained in detail. It helps learners to become familiar with words which are difficult to understand or are too technical.

- **Java** – A high-level object-oriented programming language
- **OOPS** – Object Oriented Programming System
- **Class** – Blueprint for objects
- **Object** – Instance of a class
- **Inheritance** – Acquiring properties of another class
- **Encapsulation** – Binding data and methods together
- **Polymorphism** – Many forms of one function
- **Abstraction** – Hiding implementation details

1.7. ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Check Your Progress 1

1. **Answer :** Java is a high-level, object-oriented programming language.

2. Answer : It is a style or method of writing programs.

3. Answer : Class and Object.

1.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3744	

1.9. REFERENCE

- Gosling, J. (2021). *The Java Language Specification*. Oracle Press.
- NPTEL Java Programming Notes (India)
- Oracle Java Documentation (APA 7th format)
- SWAYAM Course Materials on Java

1.10. EXERCISE

Part 1

A. Multiple choice question

1. Java is a . . . . . language.
   (a) Low-level (b) High-level
   (c) Machine-level (d) Assembly-level
2. OOPS stands for . . . . . .
   (a) Online Object Processing System
   (b) Object Oriented Programming System
   (c) Open Operating Program Structure
   (d) Object Oriented Processing Software
3. Who developed the Java programming language?
   (a) Dennis Ritchie (b) James Gosling
   (c) Bjarne Stroustrup (d) Guido van Rossum
4. Which feature of Java makes it platform independent?
   (a) Compiler (b) Interpreter
   (c) Java Virtual Machine (JVM) (d) Source Code
5. Which OOPS concept allows code reusability?
   (a) Encapsulation (b) Abstraction
   (c) Inheritance (d) Polymorphism

Answers with Explanations :

1. Ans. (b)

NOTES

Explanation : Java provides English-like syntax and hides hardware details from the programmer. Therefore, it is called a high-level programming language.

2. Ans. (b)

Explanation : OOPS means Object Oriented Programming System, where programs are designed using objects and classes to represent real-world entities.

3. Ans. (b)

Explanation : Java was developed by James Gosling at Sun Microsystems in 1995.

4. Ans. (c)

Explanation : Java programs run on the Java Virtual Machine (JVM), which allows the same program to run on different operating systems.

5. Ans. (c)

Explanation : Inheritance allows one class to use the properties and methods of another class, which helps in reusing existing code.

B. Small answer type question

1. What is meant by a programming paradigm?
2. Define Object-Oriented Programming (OOP).
3. List any two key features of OOP.
4. What is the main advantage of using OOP concepts?
5. How is Java related to the World Wide Web (WWW)?

Part 2

C. Long answer type question (200 words)

1. Define Java.
2. What is OOPS?
3. Explain inheritance.

Part 3

D. Long answer type question (300 words)

1. Explain OOPS concepts in detail.
2. Describe the evolution and applications of Java.

UNIT – 2
FEATURES AND JAVA ENVIRONMENT

UNIT STRUCTURE

2.1 Introduction
2.2 Objective
2.3 Features of Java
2.4 Java Environment
2.5 Summary
2.6 Glossary
2.7 Answer to check your progress
2.8 Suggested readings - OER
2.9 Reference
2.10 Exercise

2.1. INTRODUCTION

In the previous unit, learners were introduced to the fundamentals of programming paradigms and the evolution of Java as a modern programming language. That discussion established why Java emerged as a powerful alternative to traditional procedural and system-dependent languages. Building upon that foundation, the present unit focuses on the core features of Java and the Java execution environment, which together form the backbone of Java-based application development. Modern software systems demand portability, reliability, security, and scalability. Applications today are expected to run on heterogeneous platforms, handle multiple tasks simultaneously, and operate safely in networked and Internet-based environments. Java was designed precisely to address these requirements. Unlike earlier languages that were tightly coupled with specific operating systems, Java introduced a revolutionary execution model based on the Java Virtual Machine (JVM), enabling programs to run independently of hardware and operating systems. This shift marked a significant milestone in software engineering. Understanding Java features such as platform independence, robustness, security, and multithreading is essential for students because these features explain why Java is extensively used in enterprise systems, mobile applications, web services, and cloud platforms. These features are not isolated characteristics; instead, they work together to make Java

NOTES

reliable for real-world applications such as banking software, academic management systems, and large-scale distributed applications. Equally important is knowledge of the Java Environment, which includes the Java Development Kit (JDK), Java Runtime Environment (JRE), and Java Virtual Machine (JVM). Many beginners learn Java syntax without fully understanding how Java programs are compiled, executed, and managed internally. This unit bridges that gap by explaining how Java source code moves from development to execution, and how supporting tools and runtime systems work together.

2.2. OBJECTIVE

After completing this unit, the learner will be able to:

- **Identify** the key features of the Java programming language
- **Explain** the concept of platform independence in Java
- **Analyze** how robustness and security are achieved in Java
- **Illustrate** the role of multithreading in Java applications
- **Differentiate** between JDK, JRE, and JVM
- **Describe** the Java program execution process
- **Apply** conceptual knowledge of Java environment to basic program execution

2.3. FEATURES OF JAVA

Java is a modern, object-oriented programming language designed to meet the requirements of reliability, portability, security, and efficiency in software development. The features of Java are the primary reasons for its widespread acceptance in academic, industrial, and enterprise environments. Unlike traditional programming languages that were tightly coupled with specific hardware or operating systems, Java introduced a revolutionary execution model based on bytecode and virtual machines. Java programs are not directly executed by the operating system; instead, they run inside a controlled runtime environment that ensures consistent behavior across platforms. Additionally, Java emphasizes strong memory management, error handling, and secure execution, which are essential for large-scale and Internet-based applications. Features such as multithreading allow Java programs to handle multiple tasks concurrently, improving responsiveness and performance. Java also avoids complex and unsafe programming constructs, thereby reducing programming errors and system crashes. Together, these features make Java a robust, scalable, and secure language suitable for applications ranging from small desktop utilities to large, distributed enterprise systems. Understanding Java features helps learners appreciate why Java remains relevant even decades after its introduction and why it is a preferred language for web servers, mobile applications, cloud platforms, and academic projects.

2.3.1. PLATFORM INDEPENDENCE

Platform independence is one of the most significant and distinguishing features of Java. It refers to Java's ability to execute the same program on different operating systems and hardware platforms without requiring any modification. In traditional programming languages such as C or C++, source code is compiled into machine-specific instructions, which restricts program execution to a particular platform. Java overcomes this limitation by introducing an intermediate form of code known as bytecode.

When a Java program is compiled using the Java compiler, it is converted into bytecode rather than platform-specific machine code. This bytecode is executed by the Java Virtual Machine (JVM), which acts as an abstract computing machine. Since JVM implementations are available for different operating systems such as Windows, Linux, and macOS, the same Java bytecode can run on all these platforms. This concept is commonly summarized as "Write Once, Run Anywhere (WORA)". Platform independence significantly reduces development and maintenance costs, as developers do not need to write separate versions of the same application for different systems. It also ensures uniform behavior of applications across platforms, which is especially important in networked and Internet-based environments. For example, a Java-based university management system developed on one platform can be deployed on another platform without any changes. This feature also supports software portability in cloud computing, where applications may run on different server environments. Thus, platform independence makes Java highly adaptable, flexible, and suitable for modern distributed computing systems.

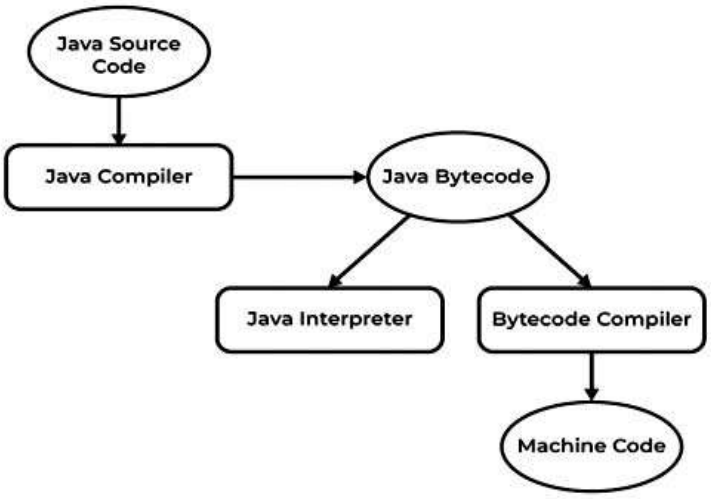


Fig. 2.3.1 : Java Platform Independence

2.3.2. ROBUSTNESS

Robustness refers to the ability of a programming language to handle errors effectively and maintain reliable execution even under unexpected conditions. Java is designed as a robust language by

NOTES

eliminating many of the error-prone features found in earlier programming languages. One of the major contributors to Java's robustness is its automatic memory management system, known as garbage collection. In Java, programmers are not required to manually allocate or deallocate memory. The garbage collector automatically identifies unused objects and frees memory, reducing the risk of memory leaks and system crashes. This feature ensures greater stability in long-running applications such as servers and enterprise systems. Java also enforces strong type checking, which detects errors at compile time rather than allowing them to occur at runtime. This reduces the likelihood of unpredictable program behavior. Additionally, Java does not support pointers, which are a common source of errors in other languages. By restricting direct memory access, Java prevents unauthorized manipulation of memory locations. Another key element of robustness is Java's exception handling mechanism. Java provides structured exception handling using try, catch, finally, and throw constructs. This allows programmers to anticipate runtime errors and handle them gracefully instead of allowing the program to terminate abruptly. As a result, Java applications can recover from errors such as invalid input, file access failures, or network interruptions. These characteristics make Java reliable for mission-critical systems such as banking applications, reservation systems, and academic management software.

2.3.3. SECURITY

Security is a fundamental requirement for software systems operating in networked and Internet-based environments. Java incorporates a comprehensive security model that protects systems from malicious code and unauthorized access. Unlike traditional languages, Java programs execute within a controlled runtime environment, which acts as a protective layer between the application and the operating system. This design prevents Java programs from directly accessing system resources without permission. Java's security begins with bytecode verification, which checks compiled code for illegal operations before execution. The class loader mechanism ensures that classes are loaded from trusted sources and prevents unauthorized replacement of core Java classes. Additionally, the security manager enforces access control policies, restricting operations such as file access, network connections, and system property modifications.

Java also supports secure communication through encryption and authentication libraries, making it suitable for developing secure web applications and online transaction systems. Features such as sandboxing allow Java applications to run in restricted environments, minimizing security risks. This robust security framework makes Java a preferred choice for applications involving sensitive data, including e-commerce platforms, e-governance portals, and financial

systems. Java's emphasis on security ensures that applications remain safe, reliable, and resistant to external threats.

2.3.4. MULTITHREADING

Multithreading is the capability of a programming language to execute multiple tasks concurrently within a single program. Java provides built-in support for multithreading, allowing developers to create highly responsive and efficient applications. A thread is the smallest unit of execution within a process, and multiple threads can run simultaneously, sharing the same resources. Java's multithreading model enables better CPU utilization and improved application performance. In Java, multithreading is implemented using the Thread class and the Runnable interface. Java manages thread scheduling, synchronization, and communication, reducing complexity for programmers. Multithreading is particularly useful in applications such as web servers, where multiple client requests must be processed simultaneously. It is also essential for graphical user interface (GUI) applications, where background tasks must run without freezing the user interface. Java provides synchronization mechanisms to avoid conflicts when multiple threads access shared resources. This ensures data consistency and prevents errors such as race conditions. By supporting multithreading at the language level, Java simplifies concurrent programming and enhances system responsiveness. As a result, Java is widely used in real-time systems, distributed computing, and high-performance enterprise applications.

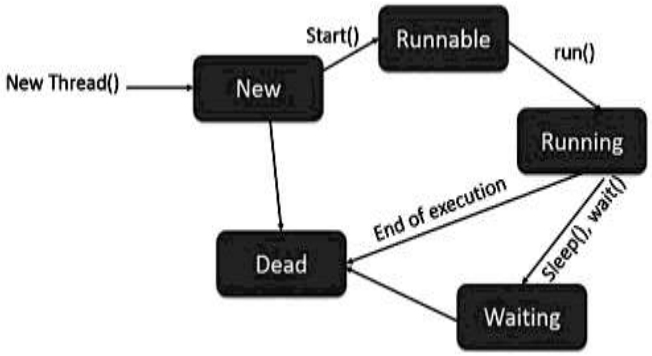


Fig. 2.3.4 (a): Java Multithreading Concept.

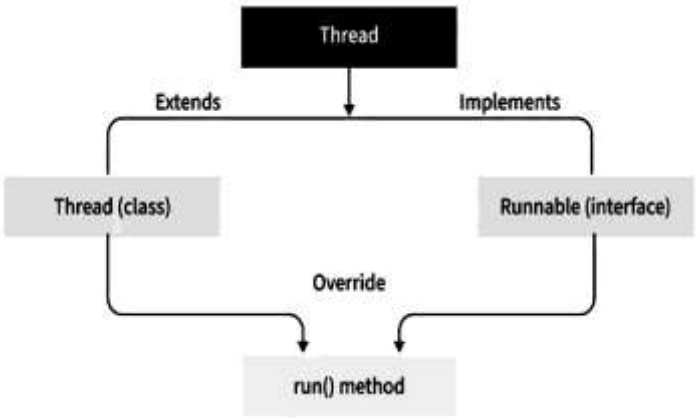


Fig. 2.3.4 (b) : Java Multithreading Concept

NOTES

2.4. JAVA ENVIRONMENT

The Java environment refers to the complete set of tools, libraries, and runtime components required to develop and execute Java programs. Unlike languages that rely solely on the operating system for execution, Java provides its own controlled environment that ensures portability, security, and reliability. The Java environment supports the entire software development life cycle, from writing and compiling code to executing and managing applications. It includes development tools, runtime libraries, and virtual machine components that work together to provide a seamless execution experience. Understanding the Java environment helps learners visualize how Java programs move from source code to execution and how platform independence is achieved. The Java environment also simplifies application deployment and maintenance, making Java suitable for enterprise and web-based systems.

2.4.1. JAVA SUPPORT SYSTEMS

Java support systems include all the auxiliary components that assist programmers during application development and execution. These systems consist of development tools, standard libraries, documentation, and runtime services. Java provides a rich set of Application Programming Interfaces (APIs) that support tasks such as file handling, networking, database connectivity, and graphical user interface creation. These APIs reduce development time and promote code reuse. In addition to APIs, Java offers debugging and monitoring tools that help developers identify and resolve errors efficiently. Integrated development environments (IDEs) such as Eclipse and NetBeans further enhance productivity by providing code editors, debuggers, and build tools. Java support systems also include runtime services such as memory management, thread scheduling, and security enforcement. These systems ensure smooth program execution and reduce the burden on developers. Overall, Java support systems play a crucial role in making Java a developer-friendly and reliable programming platform.

2.4.2. JAVA DEVELOPMENT KIT (JDK)

The Java Development Kit (JDK) is a comprehensive software package required for developing Java applications. It includes the Java compiler, runtime tools, libraries, and documentation. The compiler (javac) converts Java source code into bytecode, while other tools assist in debugging, testing, and documentation generation. JDK also includes utilities such as javadoc for creating API documentation and jar for packaging applications. JDK is essential for programmers because it provides everything needed to write, compile, and test Java programs. Without JDK, developers cannot create executable Java applications. JDK also includes the Java Runtime Environment (JRE), making it a complete development solution. Understanding JDK helps learners distinguish between development and execution environments and prepares them for professional Java development.

2.4.3. JAVA RUNTIME ENVIRONMENT (JRE) AND JVM

The Java Runtime Environment (JRE) provides the necessary environment to run Java applications. It includes the Java Virtual Machine (JVM), core libraries, and supporting files. The JVM is the heart of the Java environment and is responsible for executing bytecode, managing memory, and enforcing security policies. When a Java program runs, the JVM interprets or compiles bytecode into machine instructions specific to the underlying system. JRE ensures that Java programs run consistently across platforms by abstracting hardware and operating system details. It also manages garbage collection and thread execution, improving application reliability and performance. While JDK is required for development, JRE is sufficient for running Java applications. Together, JRE and JVM form the foundation of Java's execution model and enable platform independence.

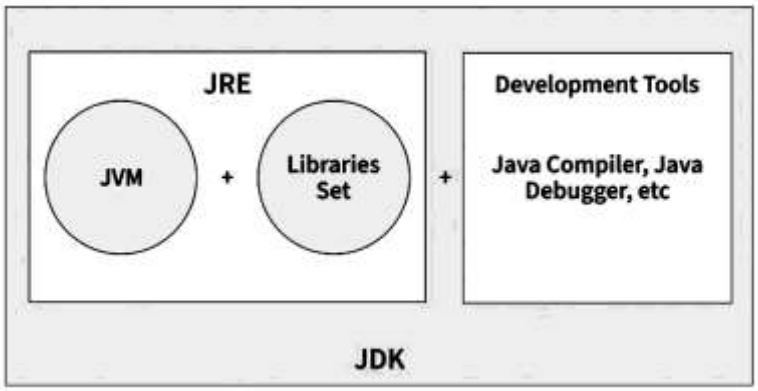


Fig. 2.4.3: Java Environment (JDK–JRE–JVM Relationship).

Activity 2.1 :

Explore the key features of Java and install the JDK on your system. Write a short note describing the Java environment and its components.

☑ Check Your Progress 2.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. Why Java is platform independent?

. . . . . .

. . . . . .

. . . . . .

2. Used of bytecode?

. . . . . .

. . . . . .

. . . . . .

NOTES

3. Concepts of Multithreading.

. . . . . .

. . . . . .

. . . . . .

2.5. SUMMARY

Java is a platform-independent programming language based on bytecode execution. The JVM enables Java programs to run on multiple operating systems.

Robustness in Java is achieved through garbage collection and exception handling. Java includes strong security mechanisms to protect system resources. Multithreading allows efficient execution of multiple tasks. JDK is required for Java application development. JRE provides the runtime environment for executing Java programs. JVM is responsible for bytecode execution and memory management.

Java features make it suitable for enterprise, web, and mobile applications. Understanding Java environment is essential for program execution.

2.6. GLOSSARY

- **Bytecode –** Intermediate code generated by Java compiler.
- **JVM –** Virtual machine that executes Java bytecode.
- **JDK –** Development toolkit for Java programmers.
- **JRE –** Runtime environment for Java applications.
- **Garbage Collection –** Automatic memory management process.
- **Multithreading –** Execution of multiple threads concurrently.
- **Platform Independence –** Ability to run programs on any OS.
- **Security Manager –** Controls access to system resources.
- **Class Loader –** Loads Java classes into memory.
- **Exception Handling –** Mechanism to manage runtime errors.

2.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** Java is platform independent because of bytecode and JVM.
2. **Answer :** JVM executes bytecode and manages runtime resources.
3. **Answer :** Multithreading improves performance and responsiveness.

2.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3743	

2.9. REFERENCES

- Balagurusamy, E. (2020). *Programming with Java*. McGraw-Hill India.
- Schildt, H. (2022). *Java: The Complete Reference*. McGraw-Hill.
- Deitel, P., & Deitel, H. (2021). *Java How to Program*. Pearson.
- Liang, Y. D. (2020). *Introduction to Java Programming*. Pearson.
- Oracle. (2023). *Java Documentation*. Oracle Corporation.

2.10. EXERCISE

Part 1

A. Multiple Choice Questions (MCQs)

1. Which feature of Java enables a program to run on different platforms without modification?
   (a) Multithreading (b) Robustness
   (c) Platform Independence (d) Security
2. Which component of Java executes bytecode instructions?
   (a) JDK (b) JRE
   (c) JVM (d) Compiler
3. Which Java feature helps in automatic memory management?
   (a) Multithreading (b) Garbage Collection
   (c) Bytecode (d) Class Loader
4. Which of the following is required to develop Java applications?
   (a) JVM (b) JRE
   (c) JDK (d) Browser
5. Which feature allows Java programs to perform multiple tasks simultaneously?
   (a) Security (b) Robustness
   (c) Platform Independence (d) Multithreading

Answers with Explanations :

1. **Ans. (c)**

NOTES

Explanation : Java programs are compiled into bytecode, which can be executed on any system having a JVM, making Java platform independent.

2. **Ans. (c)**

   Explanation : The Java Virtual Machine (JVM) is responsible for executing Java bytecode and managing runtime resources.

3. **Ans. (b)**

   Explanation : Garbage collection automatically frees unused memory, improving application reliability.

4. **Ans. (c)**

   Explanation : JDK includes the compiler, tools, and libraries needed for Java program development.

5. **Ans. (d)**

   Explanation : Multithreading enables concurrent execution of multiple threads within a program.

B. Short Answer Questions (50–100 words.)

1. What is platform independence in Java?
2. Explain the role of garbage collection in Java.
3. What is the role of the Java Virtual Machine (JVM)?
4. Differentiate between JDK and JRE.
5. Why is multithreading important in Java?

Part 2

C. Long Answer Questions (200 words)

1. Explain the major features of Java with suitable examples.
2. Describe the Java Environment and explain the role of JDK, JRE, and JVM.

Part 3

D. Extended Long Answer Questions (300 words)

1. Discuss in detail how Java achieves platform independence, robustness, and security.
2. Explain the complete life cycle of a Java program, from source code creation to execution.

UNIT – 3
JAVA LANGUAGE BASICS

UNIT STRUCTURE

3.1 Introduction
3.2 Objective
3.3 Java vs C++
3.4 Structure of Java Program
3.5 Summary
3.6 Glossary
3.7 Answer to check your progress
3.8 Suggested readings - OER
3.9 Reference
3.10 Exercise

3.1. INTRODUCTION

In the previous unit, learners studied the features of Java and the Java environment, which explained why Java is a reliable, secure, and platform-independent programming language. Building on that foundation, this unit focuses on a comparative understanding of Java with C++, followed by a detailed explanation of the structure of a Java program. This progression is important because many students entering Java programming already possess basic knowledge of C or C++. Understanding how Java differs from C++ helps learners' transition smoothly and avoid common conceptual errors. Java and C++ are both object-oriented programming languages, but they were designed with different goals. C++ evolved as an extension of the C language and emphasizes performance and system-level programming. Java, on the other hand, was designed to be simpler, safer, and more suitable for network-based and platform-independent applications. Comparing Java and C++ helps learners appreciate Java's design philosophy, especially its focus on memory safety, automatic resource management, and security. After understanding the differences between Java and C++, learners must know how a Java program is organized and executed. Unlike procedural programs, Java programs follow a structured format centered around classes and methods. The main () method acts as the entry point of execution, and all program statements are executed in a predefined flow controlled by the Java Virtual Machine (JVM). Understanding program structure is essential

NOTES

before writing even the simplest Java application. This unit also prepares learners for upcoming topics such as data types, control statements, object-oriented concepts, and exception handling. A clear understanding of Java program structure ensures that learners can read, write, and debug Java programs effectively. Overall, this unit bridges the gap between conceptual knowledge of Java features and practical program development skills, making it a crucial component of the Java programming curriculum.

3.2. OBJECTIVES

After completing this unit, the learner will be able to:

- Identify key differences between Java and C++
- Compare syntax and memory management techniques in Java and C++
- Analyze performance characteristics of Java programs
- Explain the structure of a Java program
- Illustrate the role of the main () method
- Describe the execution flow of a Java program
- Apply Java program structure to write simple programs

3.3. JAVA VS C++

Java and C++ are both widely used object-oriented programming languages, but they differ significantly in design philosophy, execution model, and application focus. C++ evolved as an extension of the C language and is primarily designed for system-level programming, where performance and direct hardware control are critical. Java, in contrast, was designed to be simpler, safer, and more portable, especially for networked and Internet-based applications. Java emphasizes platform independence through bytecode execution on the Java Virtual Machine, whereas C++ programs are compiled directly into machine code specific to the target platform. Java also enforces stricter object-oriented principles by requiring all executable code to reside within classes, while C++ allows both procedural and object-oriented styles. Another major difference lies in memory handling and security: Java avoids unsafe features such as pointers and manual memory deallocation, which are allowed in C++. This makes Java more reliable for large and long-running applications. Understanding the differences between Java and C++ helps learners transition from C/C++ to Java smoothly and avoid common conceptual mistakes, especially in memory management, program structure, and execution behavior.

3.3.1. SYNTAX COMPARISON

At a syntactic level, Java and C++ share many similarities, including the use of classes, objects, loops, conditional statements, and method definitions. This similarity exists because Java borrowed much of its

basic syntax from C and C++ to make it easier for programmers to learn. However, Java syntax is intentionally more restrictive and simplified. Java does not support multiple inheritance through classes, pointer arithmetic, operator overloading (except + for strings), or preprocessor directives such as macros and header files. In C++, header files are used to declare functions and classes separately from implementation, whereas Java uses packages and import statements, reducing complexity and improving code readability. Java also enforces strong type checking and requires explicit exception handling, which improves program safety. Another important syntactic difference is that Java does not allow standalone functions; all functions must be part of a class. In contrast, C++ allows global functions. Java also eliminates ambiguity by not supporting features like goto. These syntactic restrictions reduce programming errors and make Java programs easier to maintain, especially for beginners and large development teams.

3.3.2. Memory Management

Memory management refers to the process of allocating, using, and releasing memory during program execution. It is a critical aspect of programming because inefficient memory handling can lead to program crashes, poor performance, and system instability. One of the most significant differences between Java and C++ lies in the way these languages manage memory. C++ provides manual memory management, giving programmers direct control over memory allocation and deallocation, while Java follows an automatic memory management approach handled by the Java Virtual Machine (JVM). This fundamental difference reflects the contrasting design philosophies of the two languages: C++ prioritizes performance and low-level control, whereas Java emphasizes safety, reliability, and ease of use. In C++, memory management is largely the responsibility of the programmer. Memory can be allocated either statically or dynamically. Static memory allocation occurs at compile time, while dynamic memory allocation occurs at runtime using pointers and operators such as new and delete. While this approach allows efficient use of memory and fine-grained control over system resources, it also introduces several risks. Improper use of pointers can result in memory leaks, dangling pointers, and buffer overflows. A memory leak occurs when allocated memory is not released after use, causing gradual exhaustion of system memory. Dangling pointers occur when memory is freed, but the pointer still refers to that memory location, leading to unpredictable behavior. These issues are common in large C++ applications and are difficult to debug, especially in long-running systems such as servers.

Java addresses these challenges by eliminating manual memory management. In Java, all objects are created dynamically in heap memory, and the programmer is not required to explicitly deallocate memory. Instead, Java uses an automatic process called garbage collection. The garbage collector is a component of the JVM that

NOTES

periodically identifies objects that are no longer in use and reclaims their memory. An object becomes eligible for garbage collection when there are no active references pointing to it. This automatic approach significantly reduces the risk of memory leaks and runtime crashes, making Java programs more stable and reliable. Another major distinction between Java and C++ memory management is the handling of pointers. C++ allows direct manipulation of memory addresses through pointers, which enables high-performance system-level programming. However, pointer misuse is one of the most common sources of serious programming errors. Java completely removes pointer arithmetic and direct memory access. Instead, Java uses references that point to objects, but these references cannot be manipulated to access arbitrary memory locations. This design choice enhances security and robustness by preventing unauthorized or accidental memory access. Java's memory management system is closely integrated with its runtime environment. The JVM divides memory into different regions, such as stack memory, heap memory, and method area. Stack memory is used to store local variables and method calls, while heap memory is used to store objects created during program execution. The method area stores class-level data and bytecode. This structured memory model ensures efficient allocation and deallocation of resources. In contrast, C++ does not enforce such a standardized memory model, and memory organization may vary depending on the compiler and platform. Garbage collection in Java operates automatically in the background and follows various algorithms to optimize performance. Modern JVMs use generational garbage collection, which categorizes objects based on their lifespan. Short-lived objects are collected more frequently, while long-lived objects are collected less often. This approach improves efficiency and minimizes performance overhead. Although garbage collection introduces some runtime overhead, advancements such as Just-In-Time compilation and optimized garbage collectors have significantly reduced its impact. For most applications, the benefits of automatic memory management outweigh the minor performance costs. From a performance perspective, C++ memory management is faster because there is no garbage collection overhead. Programs compiled in C++ are converted directly into machine code and execute without an intermediary runtime system. Java programs, however, rely on the JVM, which manages memory dynamically.

While this can result in slightly slower execution, modern JVMs use adaptive optimization techniques to improve performance. As a result, Java performance is sufficient for most enterprise, web, and cloud-based applications. Another important aspect of memory management is safety. Java enforces strict runtime checks to prevent illegal memory access. Array bounds are checked automatically, and null reference exceptions are handled explicitly. In C++, accessing invalid memory locations may not produce immediate errors, making debugging more complex. Java's strict checks improve reliability and

make error detection easier, especially for beginners. Memory management also affects program maintainability. Java programs are generally easier to maintain because developers do not need to track memory allocation and deallocation manually. This reduces development time and minimizes bugs related to memory misuse. In contrast, C++ requires disciplined programming practices and careful memory tracking, which increases code complexity and maintenance effort. In summary, memory management represents a core philosophical difference between Java and C++. C++ offers manual memory control, which is powerful but error-prone, making it suitable for system-level and performance-critical applications.

Java provides automatic memory management through garbage collection, improving safety, reliability, and ease of development. This makes Java particularly suitable for large-scale, long-running, and network-based applications where stability and maintainability are more important than low-level memory optimization. Understanding these differences enables learners to choose the appropriate language for specific applications and to appreciate Java's design goals in comparison to C++.

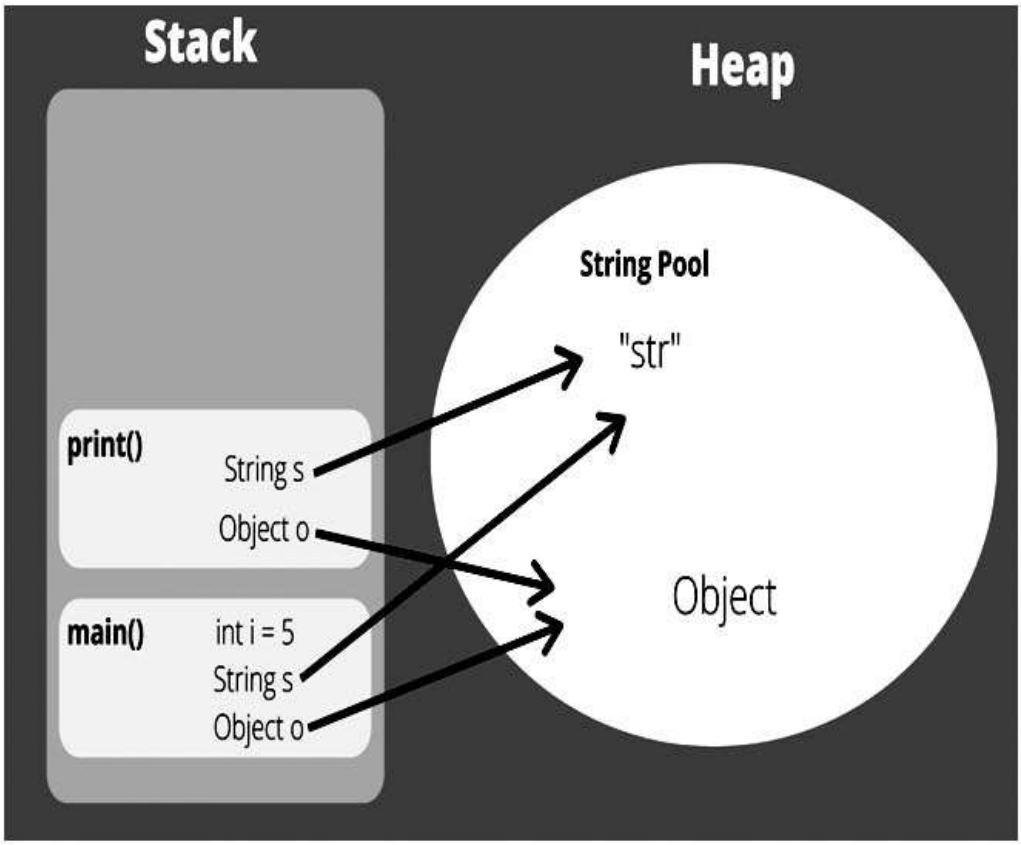


Fig. 3.3.2 : Memory Management in Java vs C++

3.3.3. PERFORMANCE DIFFERENCES

Performance is often a deciding factor when choosing between Java and C++. C++ programs generally execute faster because they are compiled directly into machine-specific code and run without an

NOTES

intermediate layer. Java programs, on the other hand, are compiled into bytecode and executed by the Java Virtual Machine, which introduces an additional level of abstraction. This can result in some performance overhead, particularly for computation-intensive tasks. However, modern JVMs use advanced optimization techniques such as Just-In-Time (JIT) compilation, which converts frequently executed bytecode into native machine code at runtime. This significantly improves Java performance and narrows the gap between Java and C++. While C++ is still preferred for system software, real-time systems, and performance-critical applications, Java provides sufficient performance for most enterprise, web, and mobile applications. Java's slightly lower performance is balanced by its portability, security, and ease of development, making it suitable for a wide range of applications.

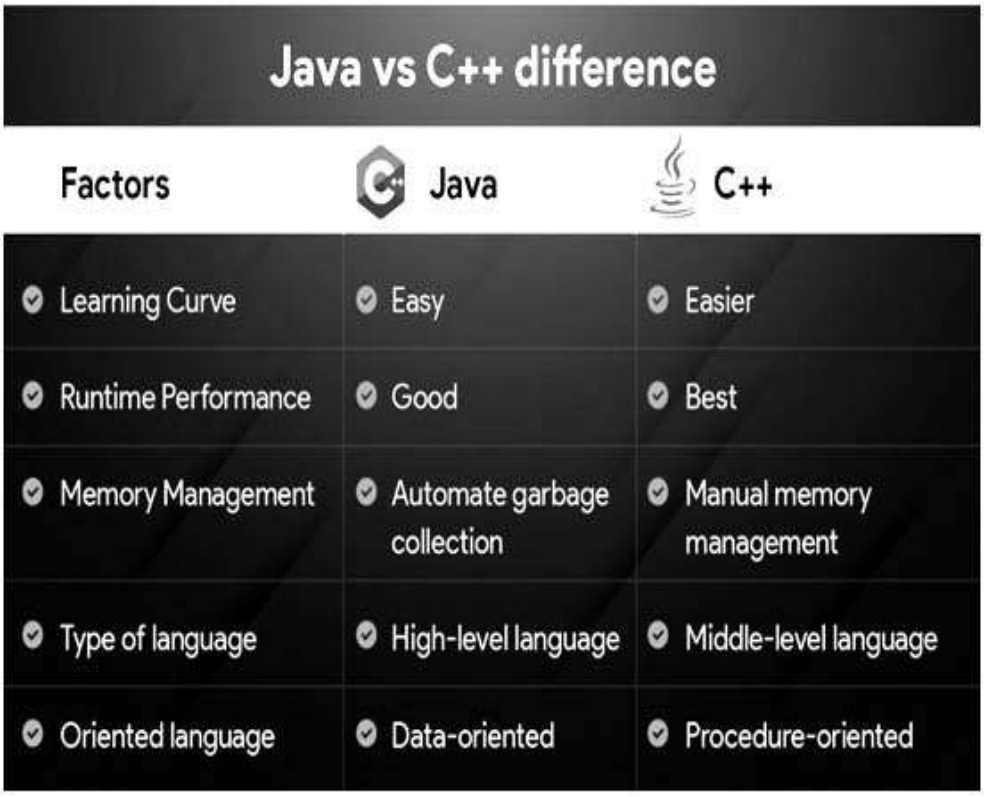


Fig. 3.3.3 (a): Java vs C++ Overview.

3.4. STRUCTURE OF JAVA PROGRAM

A Java program follows a strict and well-defined structure that ensures consistency, readability, and portability. Unlike procedural languages, Java programs are entirely class-based. Every Java program must contain at least one class definition, and program execution always begins from a specific method defined by the language specification. This structured approach enforces object-oriented principles and simplifies program organization. The structure of a Java program typically includes optional package and

import statements, a class declaration, and one or more methods. Understanding this structure is essential for writing, compiling, and executing Java programs correctly.

3.4.1. CLASS STRUCTURE

In Java, a class is the fundamental building block of a program and forms the core of Java's object-oriented design. Every Java application is structured around one or more classes, and no executable code can exist outside a class. A class acts as a blueprint or template that defines the properties (data members) and behaviors (methods) that objects created from the class will possess. This strict class-based structure distinguishes Java from procedural programming languages and ensures consistency, modularity, and reusability in program design. Understanding the structure of a Java class is essential for learners, as it lays the foundation for all advanced concepts such as objects, inheritance, polymorphism, and abstraction. A Java class definition typically begins with an optional access modifier, followed by the keyword class, the class name, and a pair of curly braces that enclose the body of the class. The class name must follow Java naming conventions and should meaningfully represent the purpose of the class. One important rule enforced by Java is that the class name must match the filename when the class is declared as public. This rule improves code organization and readability and avoids ambiguity during compilation and execution. By enforcing this structure, Java ensures that large programs remain manageable and logically organized. Within a class, various components can be defined. These include data members (fields), constructors, and methods. Data members represent the state of an object and store information relevant to the class. They can be primitive data types or references to other objects. Methods define the behavior of the class and contain executable code that operates on the data members. Constructors are special methods used to initialize objects when they are created. All these elements are enclosed within the class body, reinforcing the concept of encapsulation, which is a key principle of object-oriented programming. Java class structure also supports the use of access specifiers such as public, private, protected, and default access. These specifiers control the visibility and accessibility of class members. For example, private data members cannot be accessed directly from outside the class, ensuring data security and integrity. This controlled access mechanism allows developers to hide internal implementation details and expose only what is necessary, thereby improving program maintainability and reducing the risk of unintended interference. Another important aspect of class structure is the distinction between instance members and static members. Instance members belong to individual objects created from the class, while static members belong to the class itself and are shared among all objects. Static members

NOTES

are accessed using the class name rather than object references. This distinction allows efficient use of memory and supports logical grouping of related functionality. For example, utility methods that do not depend on object state are often declared static.

The Java class structure also supports method overloading, where multiple methods within the same class can have the same name but different parameter lists. This enhances code readability and flexibility by allowing logically similar operations to be grouped under a single method name. Additionally, Java classes can contain nested classes, which are classes defined within other classes. Nested classes help logically group related functionality and improve encapsulation in complex programs. From a program execution perspective, the Java compiler processes class definitions to generate bytecode, which is then executed by the Java Virtual Machine. Each class is compiled separately, and the JVM loads classes dynamically at runtime as needed. This dynamic class loading mechanism allows Java programs to be flexible and extensible. For example, additional classes can be loaded without recompiling the entire program, which is especially useful in large applications and frameworks. In practical applications, Java class structure promotes modular programming.

Large programs are divided into multiple classes, each responsible for a specific task. This division simplifies debugging, testing, and maintenance. Changes made to one class do not necessarily affect other classes, provided the interface remains unchanged. This modularity is one of the reasons Java is widely used in enterprise-level software development, where applications may consist of hundreds or thousands of classes. For beginners, understanding class structure also clarifies why Java programs appear more verbose than programs written in procedural languages. This verbosity is intentional and contributes to clarity, consistency, and long-term maintainability. Once learners become familiar with class structure, they can easily extend their knowledge to advanced object-oriented concepts such as inheritance, where one class acquires properties of another, and polymorphism, where methods behave differently based on object type. In summary, the class structure in Java defines how programs are organized, how data and behavior are grouped, and how objects are created and managed. It enforces object-oriented principles, supports secure and modular design, and enables scalable application development. A strong understanding of Java class structure is essential for writing correct programs, understanding program execution, and mastering advanced Java concepts. This knowledge forms the backbone of Java programming and is indispensable for both academic learning and professional software development.

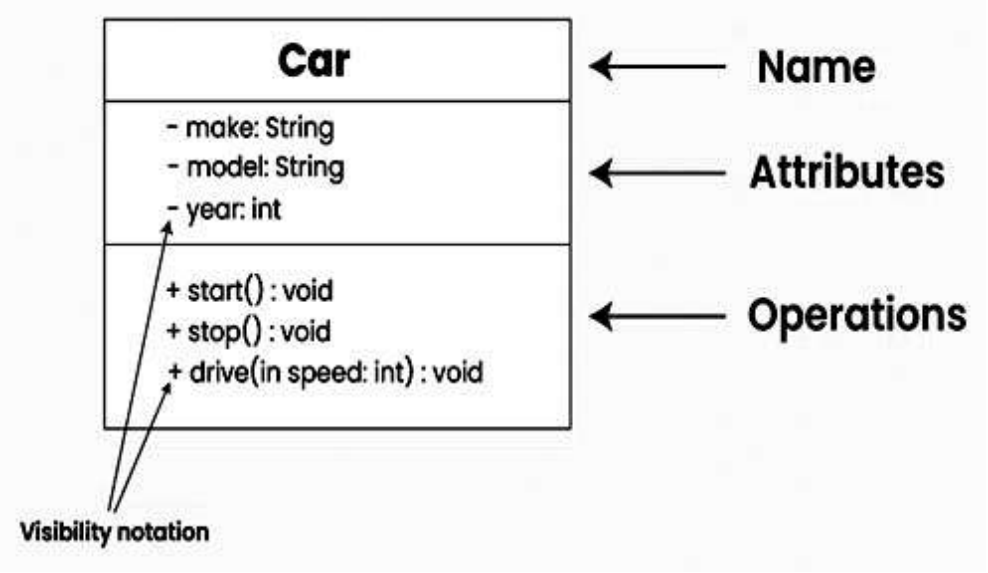


Fig. 3.4.1 : Structure of a Java Class

3.4.2. MAIN () METHOD

The main () method plays a central and indispensable role in the execution of a Java program. It serves as the starting point of execution for any standalone Java application. When a Java program is executed, the Java Virtual Machine (JVM) searches for the main () method and begins program execution from this method. Without the main () method, a Java application cannot start, making it the most critical method in the structure of a Java program. Understanding the purpose, structure, and functioning of the main () method is therefore essential for learners before they begin writing Java programs. The main () method has a fixed and predefined syntax that must be followed exactly so that the JVM can recognize it. It is declared as public to ensure that it is accessible from outside the class, since the JVM invokes it externally. The keyword static allows the JVM to call the method without creating an object of the class. This is important because no objects exist at the beginning of program execution. The return type void indicates that the main () method does not return any value to the JVM. The parameter String [] args represents an array of command-line arguments that can be passed to the program during execution. Each part of this declaration has a specific purpose and altering it incorrectly may prevent the program from running. The main () method acts as a controller of program flow. It typically contains statements that create objects, call other methods, and initiate the logial sequence of operations in the program. While small programs may contain most of their logic inside the main () method, well-structured Java programs use the main () method primarily to coordinate the execution of other methods. This approach improves modularity and readability. As programs grow larger, keeping the main () method concise becomes an important programming practice. Another important aspect of the main () method is its role in command-line argument handling. The args parameter allows users to pass input values to the program at runtime. These arguments are

NOTES

stored as strings in an array and can be processed inside the main () method. This feature is particularly useful for writing flexible and reusable programs that do not rely solely on hard-coded input. Even though many beginner-level programs may not use command-line arguments, understanding this concept prepares learners for advanced application development.

The static nature of the main () method also highlights an important concept in Java: the difference between class-level and object-level members. Since main () is static, it belongs to the class rather than to instances of the class. This allows it to access other static members directly, while non-static members must be accessed through objects. This distinction reinforces the understanding of static context, which is essential for mastering Java programming. From an execution perspective, once the JVM finds the main () method, it begins executing statements sequentially. Any errors encountered during execution, such as exceptions, are handled according to Java's exception-handling mechanism. When the main () method completes execution, the program terminates unless other non-daemon threads are still running. Thus, the lifecycle of a Java application is closely tied to the execution of the main () method. It is also important to note that Java allows only one entry point per application, which is the main () method. Although a program may contain multiple classes with main () methods, only the main () method of the class specified at runtime is executed. This flexibility allows developers to test individual classes independently while maintaining a single execution entry for the application. In summary, the main () method is the gateway through which every Java application begins execution. It connects the Java program with the JVM, controls the initial flow of execution, and provides a mechanism for receiving external input. A clear understanding of the main () method enables learners to write correct, structured, and executable Java programs. Mastery of this concept is a crucial step toward learning advanced Java programming techniques and developing real-world applications.

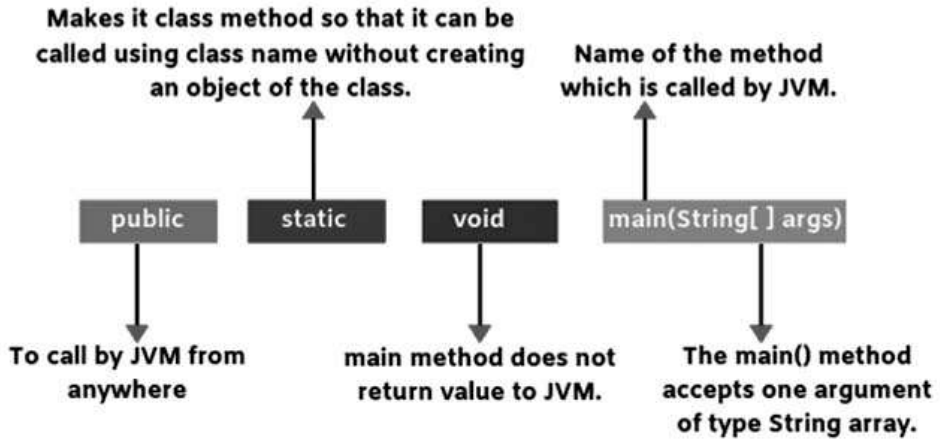


Fig. 3.4.2 (a) : Role of main () Method in Java Program Execution.

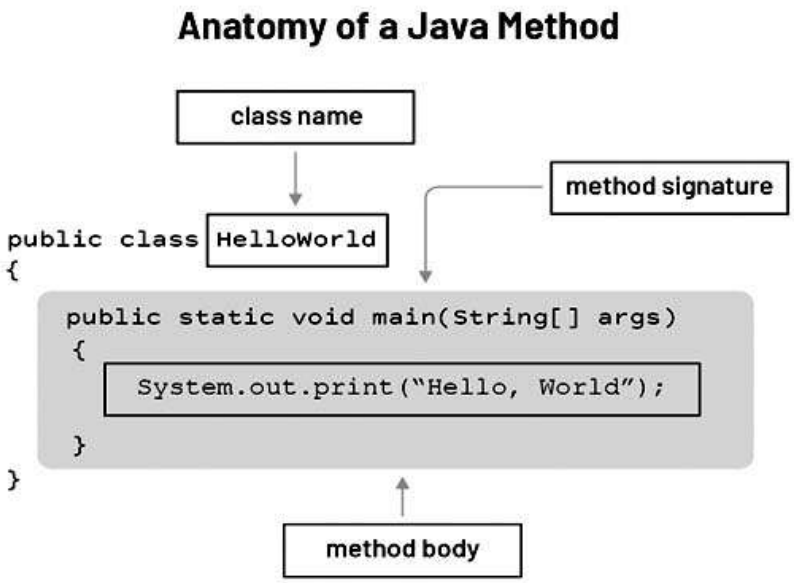


Fig. 3.4.2 (b) : Role of main () Method in Java Program Execution

3.4.3. PROGRAM EXECUTION FLOW

The execution flow of a Java program begins with the compilation of source code into bytecode using the Java compiler. This bytecode is then loaded into the Java Virtual Machine, which verifies it for correctness and security. After verification, the JVM allocates memory and begins executing instructions starting from the main () method. Execution proceeds sequentially unless altered by control structures such as loops, conditionals, or method calls. The JVM also manages runtime tasks such as memory allocation, garbage collection, and thread scheduling. Understanding program execution flow helps learners debug programs and understand runtime behavior, making it a crucial concept in Java programming.

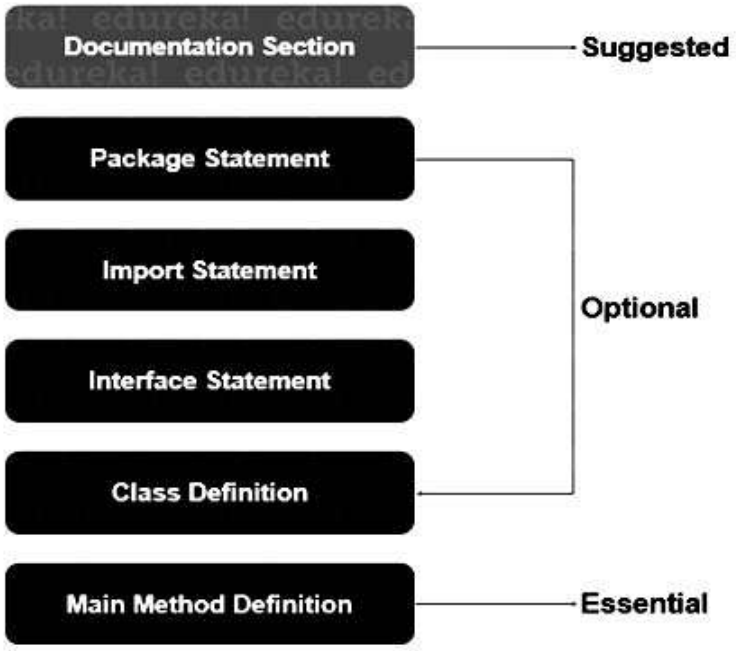


Fig. 3.4.3 : Structure and Execution Flow of Java Program.

NOTES

Activity 3.1.

Write a simple Java program to display your name and course details. Identify and label different parts of the program such as class, method, and statements.

☑ Check Your Progress 3.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. Why is Java considered platform independent?

. . . . . .

. . . . . .

. . . . . .

2. State two key differences between Java and C++.

. . . . . .

. . . . . .

. . . . . .

3. What is the role of the main () method in Java?

. . . . . .

. . . . . .

. . . . . .

3.5. SUMMARY

Java and C++ are both powerful programming languages, yet they differ in design and usage. In terms of **syntax**, Java is simpler and more consistent, removing complex features like pointers and multiple inheritance found in C++. This makes Java easier to learn and less error-prone. Regarding **memory management**, C++ uses manual memory allocation and deallocation, which offers control but increases the risk of memory leaks. Java, on the other hand, uses automatic garbage collection, making memory handling safer and more efficient for developers.

When comparing **performance**, C++ is generally faster because it compiles directly to machine code. Java programs run on the Java Virtual Machine (JVM), which introduces slight overhead but provides platform independence.

The **structure of a Java program** is organized around classes. Every program must have at least one class definition. The **main() method** serves as the entry point, written as public static void main(String[] args), from where execution begins. The **program execution flow** starts with the JVM loading the class, then calling the main method, and executing statements sequentially within it.

3.6. GLOSSARY

1. **Java :** Java is a high-level, object-oriented programming language designed to be platform independent, secure, and robust. It uses bytecode and the Java Virtual Machine to enable execution across different operating systems.
2. **C++ :** C++ is a general-purpose programming language that supports both procedural and object-oriented programming. It provides direct memory access and is commonly used for system-level and performance-critical applications.
3. **Platform Independence :** Platform independence refers to the ability of a program to run on multiple operating systems without modification. In Java, this is achieved through bytecode execution on the JVM.
4. **Bytecode :** Bytecode is an intermediate, platform-neutral code generated by the Java compiler. It is executed by the Java Virtual Machine instead of directly by the hardware.
5. **Java Virtual Machine (JVM) :** JVM is an abstract computing machine that executes Java bytecode, manages memory, controls program execution, and enforces security policies.
6. **Garbage Collection :** Garbage collection is the automatic process of identifying and freeing unused memory in Java programs, thereby preventing memory leaks and improving reliability.
7. **Class :** A class is a blueprint in Java that defines data members and methods. All Java programs are structured around classes.
8. **Object :** An object is an instance of a class that represents a real-world entity and contains state (data) and behavior (methods).
9. **main () Method :** The main () method is the entry point of a Java application. Program execution begins from this method when the program is run.
10. **Program Execution Flow :** Program execution flow refers to the order in which statements in a Java program are executed, starting from compilation and continuing through runtime execution within the JVM.

3.7. ANSWERS TO CHECK YOUR PROGRESS

1. Ans. Java is considered platform independent because Java programs are compiled into bytecode, which is executed by the Java Virtual Machine (JVM). Since JVMs are available for different operating systems, the same Java program can run on any platform without modification.

NOTES

2. Ans. Java uses automatic garbage collection for memory management and does not support pointers, which enhances safety. In contrast, C++ requires manual memory management and supports pointers, providing more control but increasing the risk of errors.

3. Ans. The main () method serves as the entry point of a Java program. When a program is executed, the JVM begins execution from this method and controls the initial flow of program instructions.

3.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3744	

3.9. REFERENCES (APA 7TH EDITION)

- Balagurusamy, E. (2020). Programming with Java. McGraw-Hill India.
- Schildt, H. (2022). Java: The Complete Reference. McGraw-Hill.
- Oracle. (2023). Java Documentation.

3.10. EXERCISE

Part 1

A. Multiple Choice Questions (MCQs)

1. Which of the following is the main reason Java is platform independent?
   (a) Use of pointers
   (b) Compilation into machine code
   (c) Use of bytecode and JVM
   (d) Manual memory management
2. Which feature is NOT supported by Java but is supported by C++?
   (a) Multithreading
   (b) Garbage collection
   (c) Pointer arithmetic
   (d) Class-based programming
3. What is the starting point of execution in a Java program?

(a) Class declaration

(b) Constructor

(c) main() method

(d) JVM

4. Which component is responsible for automatic memory management in Java?

(a) Compiler

(b) JVM

(c) JDK

(d) Operating System

5. Why is Java considered safer than C++?

(a) Faster execution

(b) Manual memory allocation

(c) Absence of pointers and strong security model

(d) Use of header files

Answers with Explanations :

1. Ans. (c)

Explanation : Java programs are compiled into bytecode, which is executed by the JVM available on different platforms.

2. Ans. (c)

Explanation : Java does not support pointers to enhance security and robustness.

3. Ans. (c)

Explanation : The JVM begins execution of a Java program from the main() method.

4. Ans. (b)

Explanation : The JVM performs garbage collection to manage memory automatically.

5. Ans. (c)

Explanation : Java avoids pointers and enforces runtime checks, improving safety.

B. Short Answer Questions

1. State two major differences between Java and C++.
2. What is the role of the main () method in Java?
3. Explain automatic memory management in Java.
4. How does Java achieve platform independence?
5. What is meant by program execution flow in Java?

NOTES

Part 2

C. Long Answer Questions

1. Compare Java and C++ with respect to syntax and memory management.
2. Explain the structure of a Java program with suitable explanation.

Part 3

D. Extended Long Answer Questions

1. Discuss Java vs C++ in terms of performance, security, and application areas.
2. Explain the complete execution flow of a Java program from compilation to execution.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | BCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC | A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

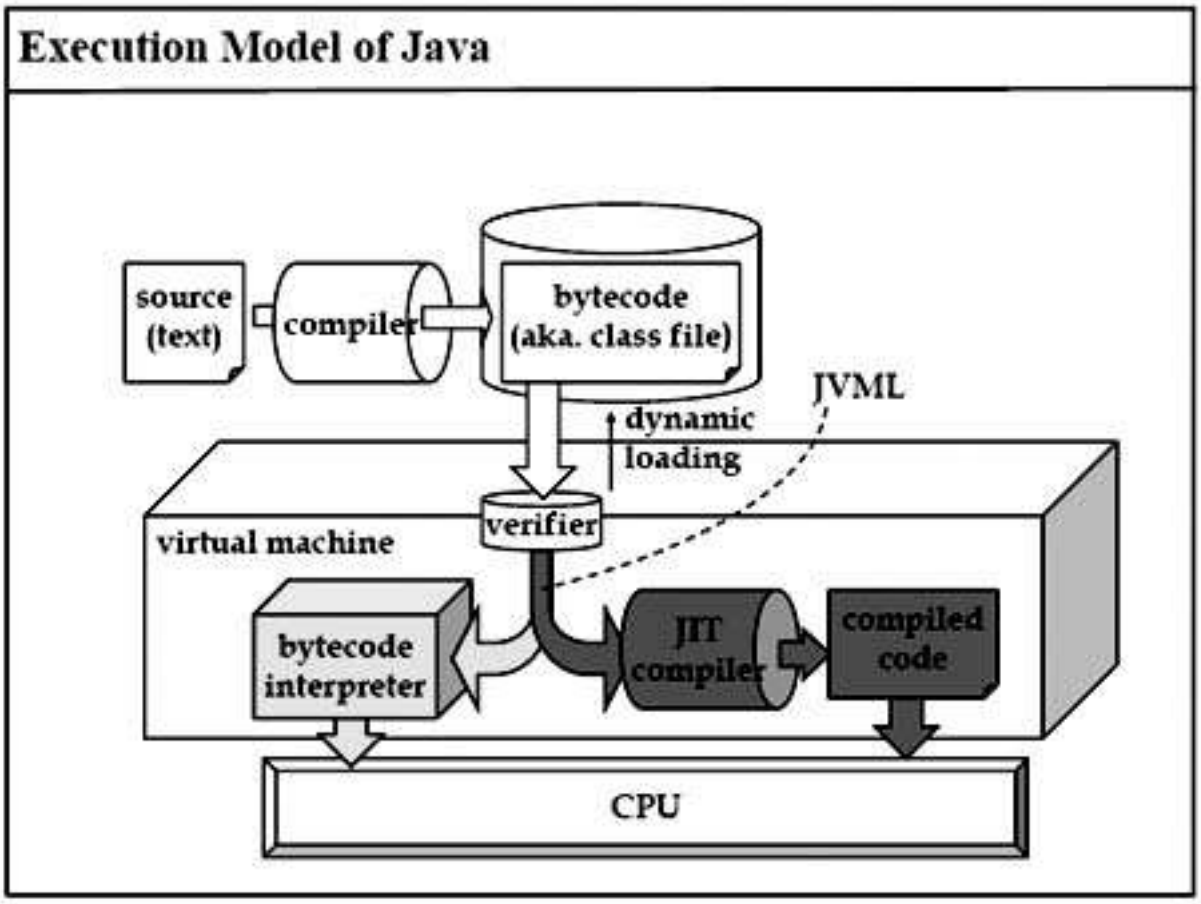
_____

_____

_____

***

Block II	JAVA EXECUTION MODEL AND DATA HANDLING	53 - 92
Unit 4	JVM and Tokens	55
Unit 5	Variables and Data Types	67
Unit 6	Variables and Data Types	81



This block introduces the foundational mechanisms that govern how Java programs are executed and how data is managed within them. It begins with an understanding of the Java Virtual Machine (JVM) and the role of tokens in program structure. Learners are then guided through variables and data types, which form the basis of data representation in Java. The block also explains type conversion and user input handling, enabling interaction between programs and users. Overall, this block lays the groundwork for writing efficient and logically sound Java programs by strengthening core programming concepts.

UNIT – 4
JVM AND TOKENS

UNIT STRUCTURE

4.1 Introduction
4.2 Objective
4.3 Java Virtual Machine
4.4 Tokens in Java
4.5 Summary
4.6 Glossary
4.7 Answer to check your progress
4.8 Suggested readings - OER
4.9 Reference
4.10 Exercise

4.1. INTRODUCTION

In the previous units, learners studied the fundamentals of Java programming, including program structure, basic syntax, and object-oriented concepts. Building on that foundation, this unit focuses on the internal working of Java programs, specifically how Java achieves platform independence and how source code is interpreted by the system. The central concept behind this mechanism is the Java Virtual Machine (JVM).

Unlike traditional programming languages such as C or C++, where the compiled program is directly executed by the operating system, Java follows a two-step execution process. First, Java source code is compiled into an intermediate representation called bytecode. This bytecode is then executed by the JVM rather than directly by the hardware. The JVM acts as a virtual execution environment that sits between the Java program and the operating system, making Java programs portable across different platforms. Th
not only for executing bytecode but also for cri
such as memory allocation, garbage collection, security enforcement, and performance optimization. Because each operating system provides its own JVM implementation that follows the same specification, the same Java program can run on Windows, Linux, or macOS without modification. This capability is popularly known as "Write Once, Run Anywhere", which has made Java one of the most

NOTES

widely used programming languages for enterprise, web, and mobile applications. In addition to understanding the JVM, this unit introduces tokens in Java, which are the smallest meaningful elements of a Java program. Every Java statement is constructed using tokens such as keywords, identifiers, literals, and separators. A clear understanding of tokens is essential for writing syntactically correct programs and for avoiding common programming errors. This unit therefore connects theoretical concepts with practical programming knowledge. It prepares learners to analyze how Java programs execute internally and helps them develop stronger coding skills by mastering Java's lexical structure. The concepts covered here also serve as a foundation for advanced topics such as memory management, exception handling, and performance tuning in Java.

4.2. OBJECTIVES

After completing this unit, the learner will be able to:

- Identify the role of the Java Virtual Machine in Java program execution
- Describe the architecture and components of the JVM
- Explain the concept and significance of Java bytecode
- Analyze the step-by-step execution process of a Java program
- Differentiate between various types of tokens in Java
- Apply rules of keywords, identifiers, and literals while writing Java programs

4.3. JAVA VIRTUAL MACHINE (JVM)

The Java Virtual Machine (JVM) is the core component of the Java platform that enables Java programs to run on different operating systems without modification. Unlike traditional programming languages that are compiled directly into machine-dependent code, Java follows a unique execution model. In this model, Java source code is first compiled into an intermediate form called bytecode, which is then executed by the JVM. The JVM acts as a virtual computer that provides a runtime environment for Java programs. It abstracts the underlying hardware and operating system, ensuring that the same Java program behaves consistently across platforms such as Windows, Linux, and macOS. This capability is famously described by the principle "Write Once, Run Anywhere (WORA)", which has been one of the major reasons for Java's widespread adoption in enterprise systems, web applications, and mobile platforms. In addition to execution, the JVM is responsible for several critical runtime services. These include dynamic memory allocation, automatic garbage collection, security enforcement, exception handling, and performance optimization. By managing these tasks internally, the JVM frees programmers from many low-level concerns

such as manual memory management, which are common in languages like C and C++. The JVM is not a physical machine; rather, it is a specification implemented differently on various platforms. Each operating system provides its own JVM implementation, but all implementations follow the same standards defined by Java. As a result, Java bytecode remains platform-independent while the JVM itself is platform-dependent.

Understanding the JVM is essential for students because it explains how Java programs execute internally, how memory is managed, and how performance optimizations occur. This knowledge becomes particularly important when developing large-scale applications, debugging runtime issues, or learning advanced Java topics.

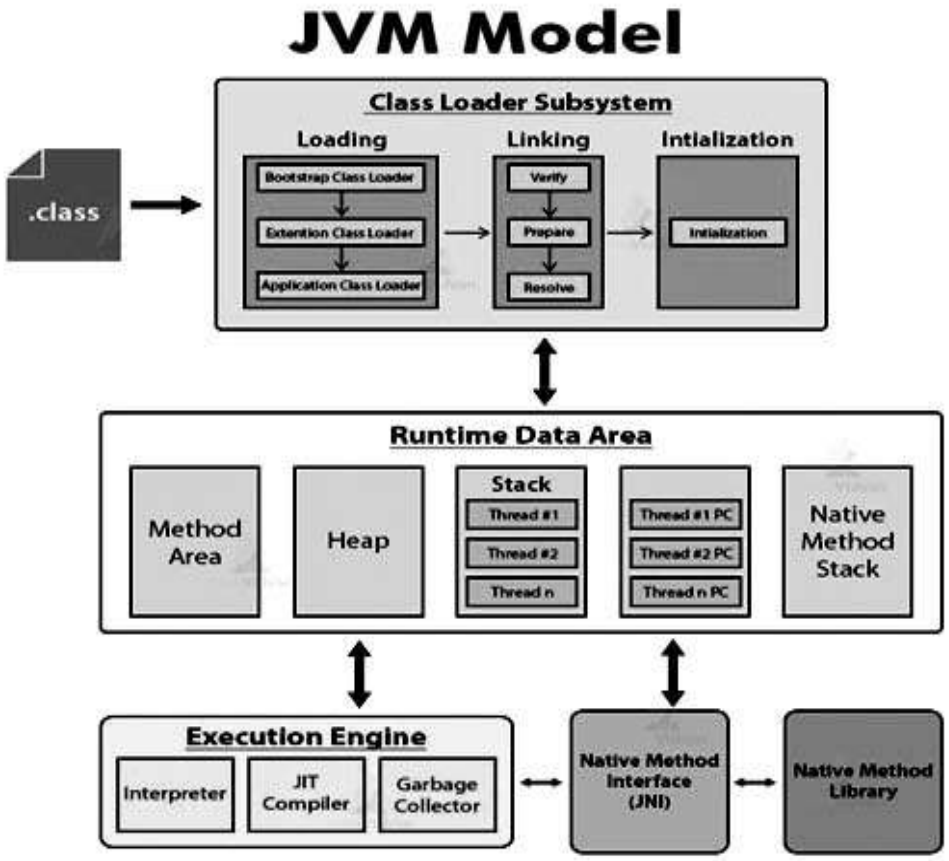


Fig. 4.1. : JVM Model.

The JVM architecture consists of several interconnected components that work together to load, verify, and execute Java bytecode. These components ensure secure and efficient execution of Java programs.

1. **Class Loader Subsystem :** The Class Loader Subsystem is responsible for loading .class files into memory. It follows the principle of lazy loading, meaning classes are loaded only when they are required.

   The class loading process consists of three phases:

   - **Loading :** Reads the .class file and creates a binary representation.

NOTES

- **Linking :** Verifies bytecode, allocates memory, and resolves symbolic references.
- **Initialization :** Executes static initializers and assigns static variables.

Types of class loaders include:

- Bootstrap Class Loader – Loads core Java libraries.
- Extension Class Loader – Loads extension libraries.
- Application Class Loader – Loads user-defined classes.

2. **Runtime Data Areas :** Runtime data areas are memory regions created by the JVM during program execution.

(a) Method Area

- Stores class metadata, method definitions, and static variables.
- Shared among all threads.

(b) Heap

- Stores objects and instance variables.
- Shared memory area.
- Managed by the garbage collector.

(c) Stack

- Stores method calls, local variables, and partial results.
- Each thread has its own stack.

(d) Program Counter (PC) Register

- Holds the address of the currently executing instruction.
- Each thread has a separate PC register.

(e) Native Method Stack

- Used for executing native (non-Java) methods.

3. **Execution Engine :** The Execution Engine executes bytecode instructions using:
   - **Interpreter –** Executes bytecode line by line.
   - **Just-In-Time (JIT) Compiler –** Converts frequently executed bytecode into native machine code for better performance.
4. **Garbage Collector :** The Garbage Collector (GC) automatically removes unused objects from heap memory. This improves memory efficiency and prevents memory leaks.

4.3.2. BYTECODE

Bytecode is the intermediate, platform-independent code generated by the Java compiler (javac). It is stored in .class files and executed by the JVM rather than directly by the operating system.

- Characteristics of Bytecode

- Platform-independent
- Compact and optimized
- Secure through verification
- Same bytecode runs on all JVMs

Importance of Bytecode

Bytecode enables Java's portability and security. Before execution, bytecode is verified to ensure that it does not perform illegal operations such as accessing unauthorized memory.

Example :

Program.java → javac → Program.class → JVM → Output

Bytecode also enables runtime optimizations through JIT compilation, making Java applications faster over time.

4.3.3. EXECUTION PROCESS :

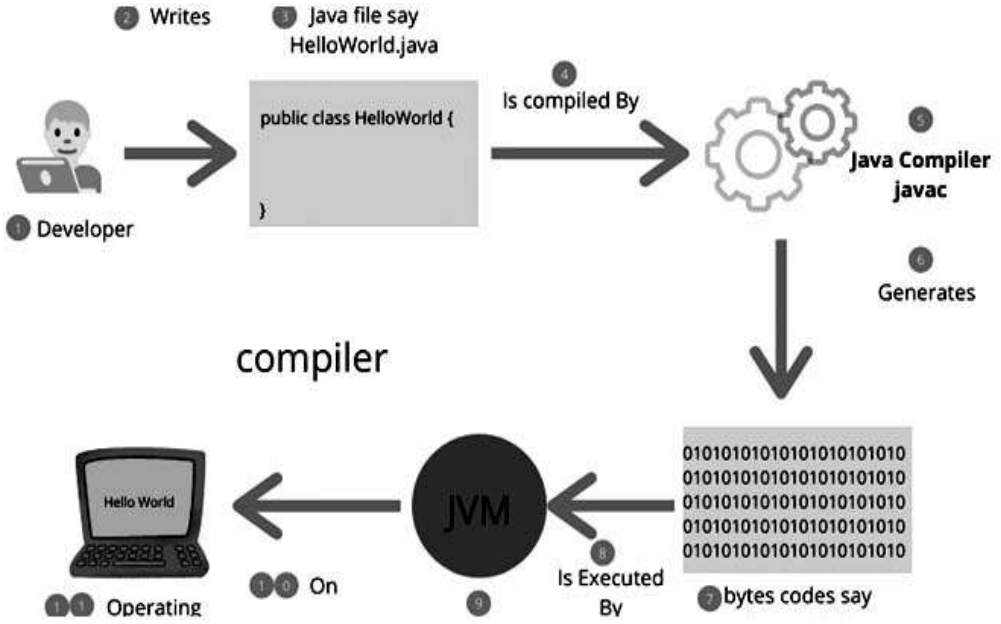


Fig. 4.2. : Execution Process

The execution of a Java program follows a structured sequence:

Step 1 : Compilation-The Java compiler converts source code (.java) into bytecode (.class).

Step 2 : Class Loading-The Class Loader loads required classes into memory.

Step 3 : Bytecode Verification-The verifier checks bytecode for safety and security.

Step 4 : Execution-The Execution Engine interprets bytecode or compiles it using JIT.

Step 5 : Garbage Collection-Unused objects are removed automatically.

This multi-stage execution ensures security, portability, and efficiency.

NOTES

4.4. TOKENS IN JAVA

Tokens are the smallest meaningful units of a Java program. The Java compiler breaks the source code into tokens to analyze program structure and syntax. Every Java statement is composed of one or more tokens. Understanding tokens helps programmers write syntactically correct programs and identify errors easily.

4.4.1. KEYWORDS

Keywords are reserved words in Java that have predefined meanings. They cannot be used as identifiers.

Examples of Java Keywords

```
class, public, static, void, int, if, else, for, return, new
```

```
Example:
public class Student {
    int marks;
}
```

Keywords define the structure and control flow of Java programs.

4.4.2. IDENTIFIERS

Identifiers are names given to variables, methods, classes, and objects.

Rules for Identifiers

- Must start with a letter, _, or $
- Cannot start with a digit
- Cannot be a keyword
- Case-sensitive

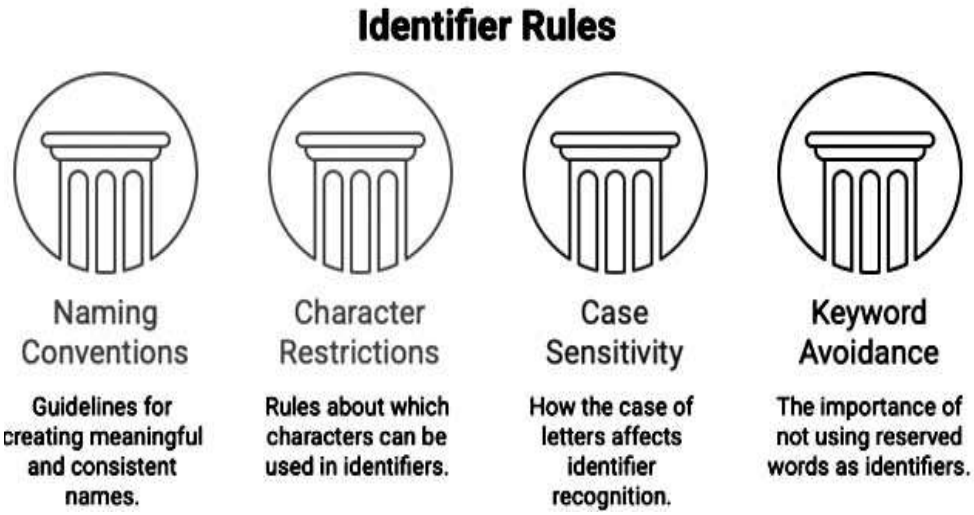


Fig. 4.3 : Idemtofoer Rules.

Examples

Valid: totalMarks, _count, StudentName

Invalid: 2marks, class, total-marks

Identifiers improve readability and maintainability of programs.

4.4.3. LITERALS

Literals represent fixed constant values in a Java program.

Types of Literals

- **Integer Literals :** 10, -25
- **Floating-Point Literals :** 3.14, 2.5f
- **Character Literals :** 'A'
- **String Literals :** "Java"
- **Boolean Literals :** true, false
- **Null Literal :** null

Example:

```
int x = 10;
String lang = "Java";
```

4.4.4. SEPARATORS

Separators are symbols used to separate logical parts of a Java program.

Common Separators

- () – Method calls
- {} – Code blocks
- [] – Arrays
- ; – Statement termination
- , – Variable separation

Example:

```
int a, b;
if (a > b) {
System.out.println(a);
}
```

Separators help the compiler understand the structure of the program.

Activity 4.1.

List the components of JVM and explain their roles briefly. Identify different Java tokens from a sample program.

NOTES

☑ **Check Your Progress 4.1.**

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What is the Java Virtual Machine (JVM)?

. . . . . .

. . . . . .

. . . . . .

2. What is Bytecode in Java? Why is it important??

. . . . . .

. . . . . .

. . . . . .

3. Explain the execution process of a Java program.

. . . . . .

. . . . . .

. . . . . .

4.5. SUMMARY

- JVM provides a runtime environment for executing Java bytecode
- Java achieves platform independence through bytecode and JVM
- JVM architecture includes class loader, runtime memory areas, execution engine, and garbage collector
- Bytecode is intermediate, secure, and platform-independent
- Java program execution follows a systematic multi-step process
- Tokens are the smallest building blocks of Java programs
- Keywords, identifiers, literals, and separators form Java syntax
- Correct usage of tokens ensures error-free programming

4.6. GLOSSARY

- **JVM** – Virtual machine that executes Java bytecode
- **Bytecode** – Intermediate compiled Java code
- **Class Loader** – Loads class files into memory
- **Heap** – Memory area for objects
- **Stack** – Memory area for method calls
- **Token** – Smallest unit of a program
- **Keyword** – Reserved word in Java
- **Identifier** – Name of a program element
- **Literal** – Constant value
- **Garbage Collection** – Automatic memory cleanup process

4.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** The Java Virtual Machine (JVM) is an abstract computing machine that enables a computer to run Java programs. It provides a runtime environment in which Java bytecode can be executed, making Java platform independent.

   JVM Architecture consists of the following main components:

   1. Class Loader Subsystem

   It loads .class files into memory when required. It has three phases: loading, linking, and initialization.

   2. Runtime Data Areas

   - **Method Area :** Stores class-level data like methods and static variables.
   - **Heap Area :** Stores objects created during program execution.
   - **Stack Area :** Stores frames for method calls, local variables, and partial results.
   - **PC Register :** Holds the address of the currently executing instruction.
   - **Native Method Stack:** Supports execution of native (non-Java) methods.

   3. Execution Engine

   Executes bytecode using the interpreter or Just-In-Time (JIT) compiler.

   4. Native Method Interface (JNI)

   Allows Java programs to interact with native libraries written in C or C++.

2. **Answer : Bytecode** is an intermediate, platform-independent code generated by the Java compiler. When a Java source file (.java) is compiled, it produces a .class file containing bytecode.

   Importance of Bytecode:

   - It makes Java platform **independent** (Write Once, Run Anywhere).
   - It is executed by the JVM, not directly by the hardware.
   - Bytecode improves security as it runs inside a controlled JVM environment.
   - It allows optimization through JIT compilation during runtime.

   Thus, bytecode acts as a bridge between Java source code and the underlying hardware.

3. **Answer :** The execution process of a Java program involves the following steps :

NOTES

1. **Writing Source Code**

   The programmer writes code in a .java file.

2. **Compilation**

   The Java compiler (javac) converts the source code into bytecode stored in a .class file.

3. **Class Loading**

   The Class Loader loads the .class file into JVM memory.

4. **Bytecode Verification**

   The JVM verifies the bytecode to ensure it is safe and follows Java security rules.

5. **Execution**

   The Execution Engine interprets bytecode line by line or compiles it using JIT for faster execution.

6. **Output Generation**

   The program produces output as specified.

4.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3745	

4.9. REFERENCES

- Balagurusamy, E. (2020). *Programming with Java*. McGraw-Hill.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Deitel, P., & Deitel, H. (2020). *Java How to Program*. Pearson.
- Oracle. (2024). *Java Documentation*.

4.10. EXERCISE

Part 1

A. Multiple Choice Questions

1. Which component of Java is responsible for executing bytecode?

   (a) Java Compiler

   (b) Operating System

   (c) Java Virtual Machine (JVM)

   (d) Assembler

2. Java bytecode is:
   (a) Platform-dependent
   (b) Machine-level code
   (c) High-level source code
   (d) Platform-independent intermediate code
3. Which memory area stores objects created in Java?
   (a) Stack
   (b) Method Area
   (c) Heap
   (d) PC Register
4. Which of the following is a Java keyword?
   (a) main
   (b) System
   (c) static
   (d) println
5. Which symbol is used as a statement terminator in Java?
   (a) ,
   (b) :
   (c) .
   (d) ;

Answers with Explanation :

1. **Ans. (c)**
   Explanation : JVM executes bytecode at runtime.
2. **Ans. (d)**
   Explanation : Bytecode is platform-independent.
3. **Ans. (c)**
   Explanation : Heap stores objects and instance variables.
4. **Ans. (c)**
   Explanation : static is a reserved Java keyword.
5. **Ans. (d)**
   Explanation : Semicolon ends Java statements.

B. Short Answer Questions (50–100 words each)

1. Define Java Virtual Machine (JVM).
2. What is Java bytecode?
3. List any four features of JVM.
4. What are tokens in Java?
5. Write rules for naming identifiers in Java.

NOTES

Part 2

C. Long Answer Questions (200 words each)

1. Explain the architecture of the Java Virtual Machine with suitable explanation.
2. Describe the step-by-step execution process of a Java program.

Part 3

D. Descriptive Answer Questions (300 words each)

1. Discuss the role of the Java Virtual Machine in achieving platform independence in Java.
2. Explain tokens in Java in detail, covering keywords, identifiers, literals, and separators with examples.

UNIT – 5
VARIABLES AND DATA TYPES

UNIT STRUCTURE

5.1 Introduction

5.2 Objective

5.3 Variables and Scope

5.4 Data Types and Constants

5.5 Summary

5.6 Glossary

5.7 Answer to check your progress

5.8 Suggested readings - OER

5.9 Reference

5.10 Exercise

5.1. INTRODUCTION

In the previous unit, learners studied the internal execution of Java programs through the Java Virtual Machine and understood the role of tokens in forming valid Java statements. Building upon that foundation, this unit focuses on how data is stored, represented, and controlled within Java programs. Variables, data types, scope, and constants are fundamental concepts that directly influence how a program behaves during execution. A variable in Java represents a named memory location used to store data values that may change while the program is running. Every Java program, regardless of its size or complexity, depends on variables to hold input data, intermediate results, and final output. However, merely declaring variables is not sufficient. Programmers must also understand where a variable is accessible, how long it exists in memory, and how its value can be modified. This concept is known as the scope of variables. Java is a strongly typed language, meaning that every variable must be declared with a specific data type. Data types determine the type of values a variable can store, the amount of memory allocated, and the operations that can be performed on that data. Proper selection of data types improves program efficiency, avoids logical errors, and ensures optimal memory usage. In many programs, certain values are not meant to change during execution, such as mathematical constants, configuration limits, or fixed reference values. For this purpose, Java provides symbolic constants, which are declared using the final

NOTES

keyword. Constants improve program readability and safety by preventing accidental modification of important values. This unit introduces learners to variable declaration, initialization, scope rules, primitive data types, constants, and data type ranges. These concepts are essential for writing correct and efficient Java programs and form the base for advanced topics such as control structures, arrays, classes, and object-oriented programming.

5.2. OBJECTIVES

After completing this unit, the learner will be able to:

- Identify different types of variables used in Java
- Declare and initialize variables correctly
- Differentiate between local, instance, and static variable scope
- Classify Java primitive data types
- Explain the use of symbolic constants in Java
- Analyze data type ranges and memory requirements
- Apply appropriate data types and constants while writing Java programs

5.3. VARIABLES AND SCOPE

In Java programming, a variable is a fundamental building block used to store data that can be accessed and modified during program execution. Every meaningful Java program relies on variables to handle user input, store calculation results, control program flow, and produce output. Without variables, programs would be limited to fixed operations and would lack flexibility. A variable can be viewed as a named container in memory. Each variable occupies a specific location in the computer's memory, and this location is referenced using the variable name.

Java requires that every variable be declared with a data type, which defines the nature of data that the variable can hold. This strong typing helps Java maintain reliability, reduce errors, and enforce disciplined programming practices. Equally important as variables themselves is the concept of scope. Scope determines where a variable can be accessed within a program and how long it exists in memory. Improper understanding of scope often leads to errors such as variable hiding, unintended value changes, and memory misuse. By carefully defining scope, Java ensures better program structure, improved readability, and safer execution. Variables and scope together form the basis for writing structured, modular, and efficient Java programs. A clear understanding of these concepts is essential before moving on to control structures, methods, arrays, and object-oriented programming.

5.3.1. VARIABLE DECLARATION

Variable declaration is the first and most essential step in using data within a Java program. In Java, a variable must be declared before it is used, which means the programmer must explicitly inform the compiler about the variable's data type and name. This requirement is a direct result of Java being a strongly typed language. When a variable is declared, the Java compiler performs two important actions. First, it reserves a specific amount of memory based on the declared data type. Second, it associates a symbolic name (the variable name) with that memory location. At this stage, the variable exists in memory, but it may not yet contain a meaningful value.

Syntax of Variable Declaration

```
dataType variableName;
```

Examples

```
int age;
double salary;
char grade;
boolean isEligible;
```

In these examples :

- int, double, char, and Boolean specify the type of data
- age, salary, grade, and is Eligible are identifiers
- Each declaration ends with a semicolon, indicating the end of the statement

Rules for Variable Declaration

To ensure correctness and avoid compilation errors, Java enforces strict rules for variable declaration:

1. A variable name must begin with a letter, underscore (_), or dollar sign ($)
2. Variable names cannot start with digits
3. Java keywords cannot be used as variable names
4. Variable names are case-sensitive
5. No two variables with the same name can exist in the same scope

Multiple Variable Declaration

Java allows multiple variables of the same data type to be declared in a single statement: `int x, y, z;`

This approach reduces code length and improves readability when variables are logically related.

Importance of Variable Declaration

Variable declaration is mandatory in Java because it:

- Helps the compiler allocate correct memory
- Prevents type-related runtime errors

NOTES

- Improves program clarity and maintainability
- Enforces disciplined programming practices

Common Errors in Declaration

Beginners often make mistakes such as:

```
int 2value;       // invalid
double class;     // invalid (keyword used)
```

Such errors are detected at compile time, making Java safer compared to loosely typed languages.

5.3.2. Initialization

Initialization is the process of assigning an initial value to a variable after it has been declared. While declaration creates the variable, initialization gives it a usable value. In Java, using a variable without initialization—especially a local variable results in a compilation error. Initialization ensures that variables contain predictable and meaningful data before they are used in calculations or output statements.

Syntax of Initialization

```
dataType variableName = value;
```

Examples

```
int age = 20;
double marks = 78.5;
char section = 'A';
boolean result = true;
```

In these examples, memory is allocated and a value is assigned in a single statement.

Declaration with Initialization

Java allows declaration and initialization to be combined:

```
int count = 10;
```

This is the most used and recommended approach, as it reduces errors caused by uninitialized variables.

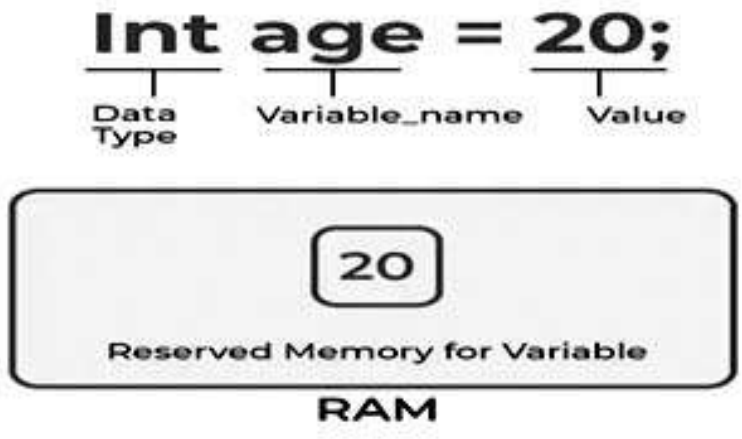


Fig. 5.1 : Declaration with Initialization

1. Explicit Initialization

The programmer assigns a value manually:

```
int total = 100;
```

2. Default Initialization

Java automatically assigns default values to:

- **Instance variables**
- **Static variables**

Examples of default values:

- int → 0
- double → 0.0
- boolean → false
- char → '\u0000'

Local variables do not receive default values and must be explicitly initialized.

Why Initialization Is Mandatory

Initialization plays a critical role in program correctness:

- Prevents unpredictable output
- Avoids use of garbage values
- Improves program reliability
- Ensures logical consistency

Example of error due to missing initialization:

```
int x;
System.out.println(x);   // Compilation error
```

Initialization During Runtime

Variables can also be initialized using expressions or user input:

```
int a = 10;
int b = a + 5;
```

This allows dynamic assignment of values and supports logical computations.

Best Practices for Initialization

- Always initialize variables at the time of declaration when possible
- Use meaningful initial values
- Avoid relying on default initialization for program logic
- Keep initialization close to declaration for better readability

Difference Between Declaration and Initialization

Declaration	Initialization
Reserves memory	Assigns value
Does not give usable data	Makes variable usable
Mandatory before use	Mandatory for computation

NOTES

5.3.3. SCOPE OF VARIABLES

Scope refers to the region of a program where a variable is visible and accessible. It also determines the lifetime of a variable, i.e., how long it exists in memory. Java defines scope to avoid ambiguity and conflicts between variables.

Types of Variable Scope in Java

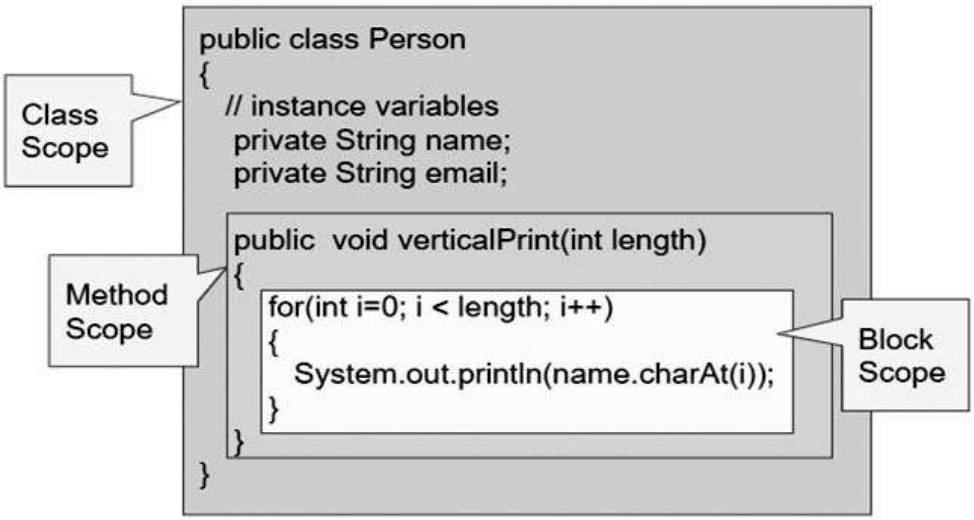


Fig. 5.2. : Types of Varibale Scope in Java.

1. **Local Variables :** Local variables are declared inside methods, constructors, or blocks.

   Characteristics:

   - Accessible only within the block
   - No default value
   - Stored in stack memory

   Example

```
void show() {
int x = 10;
System.out.println(x);
}
```

2. **Instance Variables :** Instance variables are declared inside a class but outside all methods.

   Characteristics:

   - Each object has its own copy
   - Default values are assigned
   - Stored in heap memory

   Example

```
class Student
{
int marks;
}
```

3. Static Variables: Static variables are declared using the static keyword.

Characteristics:

- Shared among all objects
- Single copy in memory
- Used for common data

Example

```
class College {
static String name = "ABC College";
```

5.4. DATA TYPES AND CONSTANTS

Data types define the kind of data a variable can store, the amount of memory allocated, and the operations permitted on that data. Java's strict data type system improves program reliability and memory efficiency. Constants represent values that remain fixed throughout program execution. They are essential when dealing with values that should not change, such as mathematical constants or system limits.

5.4.1. Primitive Data Types

Java provides eight primitive data types, which store simple values directly in memory.

List of Primitive Data Types

Data Type	Size	Use
byte	1 byte	Small integers
short	2 bytes	Medium integers
int	4 bytes	Default integer
long	8 bytes	Large integers
float	4 bytes	Decimal numbers
double	8 bytes	High-precision decimals
char	2 bytes	Single character
boolean	1 bit	true/false

Examples

```
int total = 100;
double avg = 75.25;
char grade = 'B';
boolean passed = true;
```

NOTES

5.4. Symbolic Constants

A symbolic constant is a named constant whose value cannot be changed once assigned. Java uses the final keyword to declare constants.

Syntax

```
final dataType CONSTANT_NAME = value;
```

Examples

```
final double PI = 3.14159;
final int MAX_STUDENTS = 60;
```

Advantages

- Improves readability
- Prevents accidental modification
- Simplifies maintenance
- Encourages meaningful naming

5.4.3. Data Type Ranges

In Java programming, every data type has a fixed range of values that it can store. The data type range defines the minimum and maximum values that a variable of a particular type can hold in memory. These ranges are determined by the number of bits allocated to the data type and whether the data type is signed or unsigned. Understanding data type ranges is extremely important for writing correct and efficient Java programs. When a value exceeds the allowable range of a data type, it results in overflow or underflow, which can cause incorrect results or unexpected program behavior. Therefore, programmers must select data types carefully based on the nature and size of data being processed. Java standardizes the size and range of its primitive data types, which ensures platform independence. Unlike some programming languages where data type sizes may vary depending on the system architecture, Java guarantees the same range across all platforms. This consistency makes Java programs reliable and predictable.

Concept of Signed and Unsigned Data

Most numeric data types in Java are signed, meaning they can store both positive and negative values. The only exception is the char data type, which is unsigned and stores only non-negative values.

- **Signed data types** use one bit to represent the sign (positive or negative).
- **Unsigned data types** use all bits to represent magnitude.

This distinction directly affects the range of values a data type can store.

Ranges of Integer Data Types

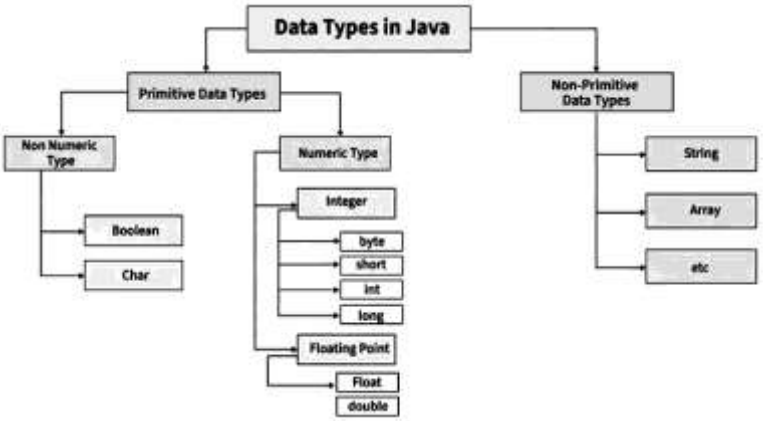


Fig. 5.3 : Data Types in Java.

Type	Size (in bits)	Range
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2^{31} to $2^{31}-1$
long	64	-2^{63} to $2^{63}-1$
float	32	1.4e-045 to 3.4e+038
double	64	4.9e-324 to 1.8e+308
char	16	0 to 65,535
boolean	1	true or false

Java provides four integer data types: byte, short, int, and long. Each differs in size and range.

1. ***byte***
   - Size: 1 byte (8 bits)
   - Range: –128 to 127

   The byte data type is useful for saving memory in large arrays where values are small.

   Example:

   byte level = 100;

2. ***short***
   - Size: 2 bytes (16 bits)
   - Range: –32,768 to 32,767

   The short type is rarely used but can be helpful when memory conservation is critical.

   Example :

   ```
   short year = 2024;
   ```

3. ***int***
   - Size: 4 bytes (32 bits)
   - Range: -2^{31} to $2^{31}-1$

   The int data type is the default and most used integer type in Java.

NOTES

Example:

int population = 1500000;

4. *long*

- Size: 8 bytes (64 bits)
- Range: -2^{63} to $2^{63}-1$

The long type is used when very large integer values are required.

Example:

```
long distance = 9876543210L;
```

Range of Character Data Type

5. *char*

- Size: 2 bytes (16 bits)
- Range: 0 to 65,535

The char data type is unsigned and is based on Unicode, allowing Java to support international characters.

Example:

```
char letter = 'A';
char symbol = '@';
```

Because char uses Unicode, it can represent characters from multiple languages.



Floating-point data types are used to store decimal and fractional values.

1. *float*

- Size: 4 bytes
- Range: Approximately $\pm 3.4 \times 10^{38}$
- Precision: Single precision

Example:

```
float temperature = 36.5f;
```

2. *double*

- Size: 8 bytes
- Range: Approximately $\pm 1.7 \times 10^{3}$

- Precision: Double precision

The double data type is the default for decimal values in Java.

Example:

```
double average = 78.98765;
```

Range of Boolean Data Type

3. *boolean*

- Size: JVM dependent
- Values: true or false

The boolean data type does not have a numeric range. It is used for logical conditions and decision-making.

Example:

```
boolean isPassed = true;
```

Importance of Understanding Data Type Ranges

Understanding data type ranges helps programmers to:

- Avoid overflow and underflow errors
- Select appropriate data types for variables
- Optimize memory usage
- Improve program performance
- Ensure accuracy in calculations

Example of Overflow:

```
byte b = 127;
b++;   // results in overflow
```

This example shows why selecting the correct data type is critical.

Best Practices Related to Data Type Ranges

- Use int for general integer calculations
- Use long for very large numbers
- Use double for precise decimal calculations
- Avoid using smaller data types unless memory optimization is required.
- Always consider the expected range of values before choosing a data type.

Activity 5.1. Variable Mapping in Business Scenarios

Objectives:

- To understand how variables function in real-world business contexts
- To differentiate between types of variables and their scope
- To build analytical thinking through application

Materials Needed:

- Notebook or digital document
- Any programming environment (optional, e.g., Python/Excel)

Activity Description:

Select a simple business scenario such as calculating employee bonuses, sales commissions, or inventory tracking.

1. Identify at least **5 variables** involved (e.g., salary, bonus rate, sales

NOTES

amount).
2. Classify them as **local or global variables** based on where they would be used.
3. Write a short pseudo-code or simple program illustrating how these variables interact.
4. Modify one variable's scope (e.g., change a global variable to local) and observe how it impacts the logic.

Reflection / Outcome:
Write a brief note (100–150 words) explaining how incorrect handling of variable scope could affect business decisions or system outputs.

☑ Check Your Progress 5.1.

1. What is a variable? Explain variable declaration and initialization in Java.
   . . . . . .
   . . . . . .
   . . . . . .
2. What is the scope of a variable? Explain different types of variable scope.?
   . . . . . .
   . . . . . .
   . . . . . .
3. What are primitive data types and symbolic constants? Explain data type ranges.
   . . . . . .
   . . . . . .
   ....

5.5. SUMMARY

The concepts of variables and scope form the foundation of any programming or data-handling task. A variable is essentially a named storage location used to hold data that can change during program execution. In business applications, variables are used to store dynamic information such as sales figures, costs, or customer data. Scope determines where a variable can be accessed within a program. Variables may have local scope, meaning they are accessible only within a specific function or block, or global scope, meaning they can be accessed throughout the program. Proper understanding of scope ensures better control over data, prevents errors, and improves program efficiency.

5.6. GLOSSARY

- **Variable:** A named storage location used to hold data that can change during program execution.
- **Scope**: The region of a program where a variable can be accessed or used.
- **Initialization:** The process of assigning an initial value to a variable at the time of its creation.
- **Data Type:** A classification that defines the type of data a variable can store, such as integer, float, or string.

- **Primitive Data Type**: Basic built-in data types that store simple values, such as integers, floats, and booleans.

5.7. ANSWER TO CHECK YOUR PROGRESS

1. **Answer :** A variable is a named memory location used to store data. Variable declaration specifies the data type and name of the variable (e.g., int a;). Initialization assigns a value to the variable (e.g., a = 10;). Declaration and initialization can also be done together, such as int a = 10;.
2. **Answer :** The scope of a variable defines where it can be accessed in a program. Types of scope include:
   - **Local variables :** Declared inside methods or blocks and accessible only within them.
   - **Instance variables :** Declared inside a class but outside methods, accessible through objects.
   - **Class (static) variables :** Declared with the static keyword and shared among all objects of the class..
3. **Answer :** Primitive data types are basic data types provided by Java, such as `int`, `float`, `double`, `char`, and `boolean`.
   Symbolic constants are fixed values defined using the final keyword (e.g., final int MAX = 100;).
   Each data type has a specific range, such as:
   - byte: -128 to 127
   - int: -2^{31} to $2^{31}-1$
   - float: approximately $\pm 3.4 \times 10^{38}$
   - double: approximately $\pm 1.7 \times 10^{308}$

   These ranges determine the limits of values that can be stored.

5.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3746	

5.9. REFERENCES (APA 7TH EDITION)

- Balagurusamy, E. (2020). *Programming with Java*. McGraw-Hill.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Deitel, P., & Deitel, H. (2020). *Java How to Program*. Pearson.
- Oracle. (2024). Java Documentation.

5.10. EXERCISE

Part 1

A. Multiple Choice Questions

1. Which keyword is used to declare a symbolic constant in Java?

   (a) static (b) const (c) final (d) define

2. Which data type is used to store decimal values with single precision?

NOTES

(a) double (b) int (c) float (d) long

3. Which variable is accessible only within the method where it is declared?

   (a) Instance variable (b) Static variable

   (c) Global variable (d) Local variable

4. What is the default integer data type in Java?

   (a) byte (b) short (c) int (d) long

5. Which of the following data types stores true or false values?

   (a) char (b) int (c) Boolean (d) byte

Answers with Explanation :

1. Ans. (c)

Explanation: The final keyword prevents the value of a variable from being modified.

2. Ans. (c)

Explanation: float stores single-precision floating-point numbers.

3. Ans. (d)

Explanation : Local variables have block-level scope.

4. Ans. (c)

Explanation : int is the most used integer type.

5. Ans. (c)

Explanation : The boolean data type stores logical values.

B. Short Answer Questions (50–100 words each)

1. Define a variable in Java.
2. What is meant by initialization of a variable?
3. Explain the scope of a variable.
4. What are symbolic constants? Why are they used?
5. List any four primitive data types in Java with examples.

Part 2

C. Long Answer Questions (200 words each)

1. Explain different types of variables in Java based on scope with suitable examples.
2. Describe Java primitive data types and explain their importance in program design.

Part 3

D. Descriptive Answer Questions (300 words each)

1. Discuss the concept of variable scope in Java. Explain local, instance, and static variables with examples.
2. Explain data types and symbolic constants in Java. Also discuss data type ranges and their significance.

UNIT – 6
TYPE CONVERSION AND INPUT

UNIT STRUCTURE

6.1 Introduction
6.2 Objective
6.3 Type Casting
6.4 Command Line Arguments
6.5 Summary
6.6 Glossary
6.7 Answer to check your progress
6.8 Suggested readings - OER
6.9 Reference
6.10 Exercise

6.1. INTRODUCTION

In the previous unit, learners developed a strong foundation in the basic building blocks of Java programming, including data handling, variables, data types, and constants. Building upon that knowledge, the present unit focuses on advancing the learner's understanding of how Java programs are structured, controlled, and logically organized to perform meaningful tasks. This unit acts as a bridge between basic syntax knowledge and practical program development.

As Java programs grow and complexity, it becomes essential to apply logical structures, rules, and mechanisms that allow programs to make decisions, repeat actions, or manage execution flow effectively. The concepts introduced in this unit help learners move beyond simple sequential execution and begin writing programs that respond dynamically to different situations.

This unit emphasizes clarity of logic, correctness of execution, and structured program design. It encourages learners to think analytically, break problems into smaller steps, and implement solutions using appropriate programming constructs. The knowledge gained here will support learners in writing efficient, readable, and maintainable Java programs.

The concepts discussed in this unit are widely used in real-world applications such as form validation, menu-driven programs, data processing systems, and interactive applications. Mastery of these ideas is crucial before progressing to advanced Java topics and object-oriented programming.

NOTES

6.2. OBJECTIVES

After completing this unit, the learner will be able to:

- **Identify** the key concepts introduced in this unit
- **Explain** the purpose and importance of these concepts in Java programming
- **Apply** the concepts to develop logically correct Java programs
- **Analyze** program behavior based on given conditions or structures
- **Differentiate** between various programming constructs covered in this unit
- **Construct** simple to moderately complex Java programs using these concepts.

6.3. TYPE CASTING

In Java programming, data is often required to move from one form to another. During program execution, values stored in variables of one data type may need to be converted into another data type so that calculations, comparisons, or assignments can be performed correctly. This process is known as type casting. Type casting refers to the conversion of a value from one data type to another. Java enforces strict rules for type conversion because it is a strongly typed language. These rules ensure that data integrity is maintained and that unintended data loss is minimized. Type casting becomes especially important when dealing with arithmetic expressions involving mixed data types, user input, method arguments, and return values.

Java supports two major forms of type casting :

1. **Implicit Type Casting (Widening Conversion)**
2. **Explicit Type Casting (Narrowing Conversion)**

Understanding these two forms is essential for writing error-free Java programs and for controlling how data behaves during execution.

Why Type Casting Is Required

Type casting is required in Java for several reasons:

- To perform operations between different data types
- To store values of one type into variables of another type
- To control precision and memory usage
- To avoid compilation errors caused by incompatible types
- To ensure correct output in arithmetic expressions

For example, adding an integer value to a floating-point value requires the compiler to convert one type into another so that the operation can be performed safely

Classification of Type Casting in Java

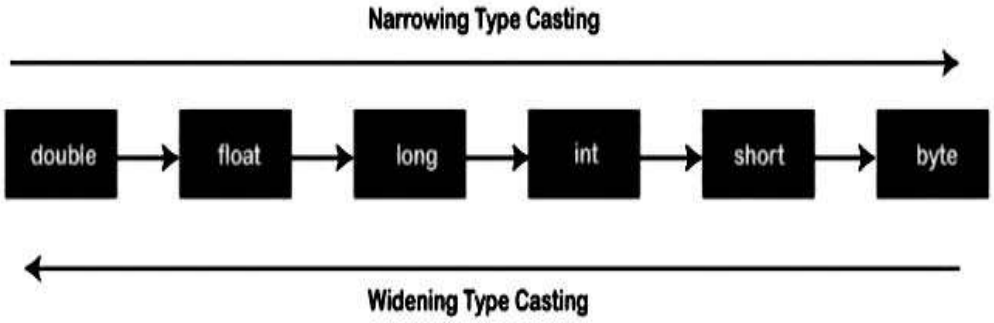


Fig. 6.1 : Type Casting in Java.

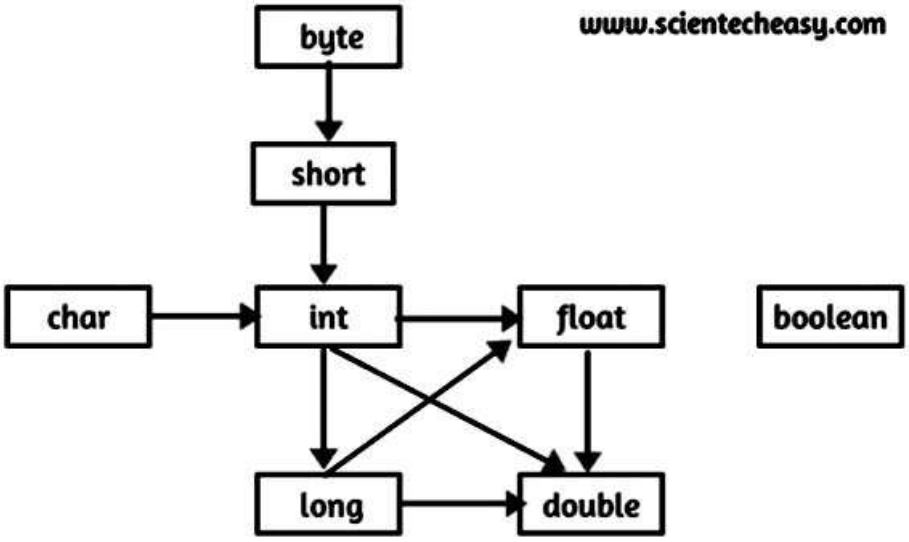


Fig. 6.2 : Automatic type conversion that Java allows.

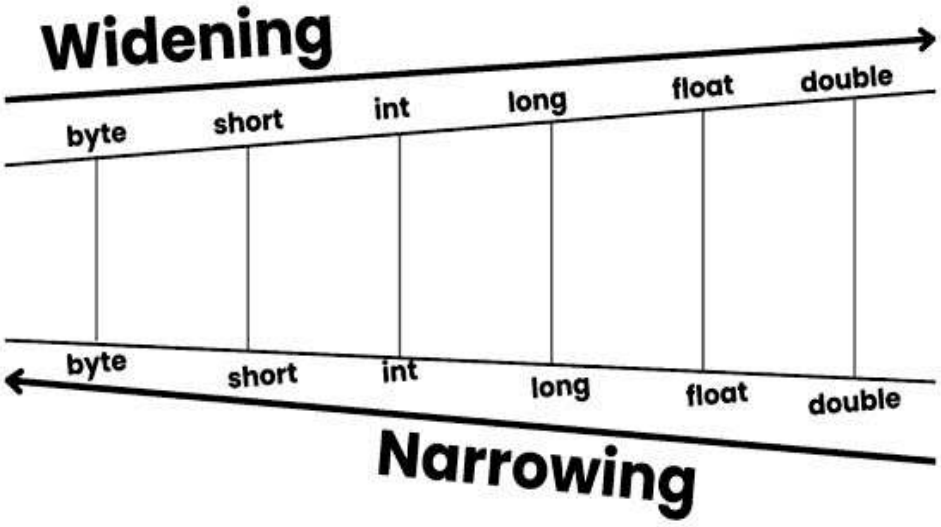


Fig. 6.3 : Java classifies type casting into two broad categories.

Java classifies type casting into two broad categories:

- Implicit Type Casting (Automatic)
- Explicit Type Casting (Manual)

Each type follows specific rules and is applied in different situations.

NOTES

6.3.1. Implicit Type Casting

Implicit type casting, also known as automatic type casting or widening conversion, occurs when a value of a smaller data type is converted into a larger data type automatically by the Java compiler. This type of casting is considered safe because there is no loss of data. In implicit type casting, the compiler itself performs the conversion without requiring any special syntax from the programmer.

Conditions for Implicit Type Casting

Implicit type casting occurs when:

- The source data type is smaller than the destination data type
- The conversion does not cause data loss
- Both data types are compatible

Order of Widening Conversion

byte → short → int → long → float → double

This sequence shows how Java safely promotes smaller data types to larger ones.

Examples of Implicit Type Casting

```
int a = 10;
double b = a;   // implicit casting
```

Here, the integer value 10 is automatically converted into 10.0.

Another example:

```
char ch = 'A';
int value = ch;
```

In this case, the Unicode value of character 'A' is stored in the integer variable.

Implicit Casting in Expressions

When an expression contains multiple data types, Java automatically promotes smaller types:

```
int x = 5;
double y = 2.5;
double result = x + y;
```

The integer x is automatically converted into a double before addition.

Advantages of Implicit Type Casting :

- Safe and error-free
- No data loss
- Simplifies code
- Improves readability
- Handled automatically by compiler

Limitations of Implicit Type Casting :

- Only works for widening conversions

- Cannot convert larger types to smaller types
- Does not give control over precision

Practical Use Cases :

- Arithmetic operations
- Method parameter passing
- Returning values from methods
- Mixed-type expressions

6.3.2. Explicit Type Casting

Explicit type casting, also known as manual type casting or narrowing conversion, occurs when a value of a larger data type is converted into a smaller data type. Since this conversion may lead to data loss, Java requires the programmer to explicitly specify the conversion. In explicit casting, the programmer takes responsibility for the conversion.

Syntax of Explicit Type Casting :

(destinationType) value;

Examples of Explicit Type Casting

```
double d = 9.7;
int i = (int) d;
```

The fractional part .7 is lost, and only 9 is stored.

Another example:

```
int x = 130;
byte b = (byte) x;
```

Here, overflow occurs because the value exceeds the byte range.

Explicit Casting in Expressions :

```
int a = 10;
int b = 3;
double result = (double) a / b;
```

Explicit casting ensures accurate division.

Possible issues:

- Loss of decimal part
- Overflow or underflow
- Unexpected output

Advantages of Explicit Type Casting

- Gives programmer full control
- Enables precision management
- Required for narrowing conversions

Risks of Explicit Type Casting

- Data loss
- Overflow/underflow

NOTES

- Logical errors if misused

Best Practices :

- Use explicit casting only when necessary
- Always check value ranges
- Avoid casting that causes overflow
- Prefer widening conversions when possible

6.4. COMMAND LINE ARGUMENTS

Command Line Arguments provide a powerful mechanism to pass data to a Java program at runtime. Instead of hard-coding values or using interactive input, command line arguments allow users to supply input when the program starts executing. This feature is widely used in real-world applications such as batch processing, automation scripts, configuration-based programs, and system utilities.

Concept of Command Line Arguments : Command line arguments are values passed to the main() method of a Java program during execution. These arguments are stored as strings and accessed using an array parameter.

Syntax of main() Method

public static void main(String[] args)

Here:

- args is an array of String
- Each argument is stored as an element of the array

6.4.1. Passing Arguments

Arguments are passed from the command prompt at the time of execution: java ProgramName arg1 arg2 arg3

Each space-separated value is treated as a separate argument.

Example Program

```
class Demo {
    public static void main(String[] args) {
        System.out.println("First argument: " +
args[0]);
    }
}
```

Execution :

java Demo Hello

Output:

First argument: Hello

Multiple Arguments Example

```
class Sum {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        System.out.println("Sum = " + (a + b));
    }
}
```

6.4.2. Accessing Arguments

Arguments are accessed using array indexing:

```
args[0], args[1], args[2]
```

Handling Missing Arguments

```
if (args.length == 0)
{
System.out.println("No arguments passed");
}
```

Converting Arguments to Other Data Types

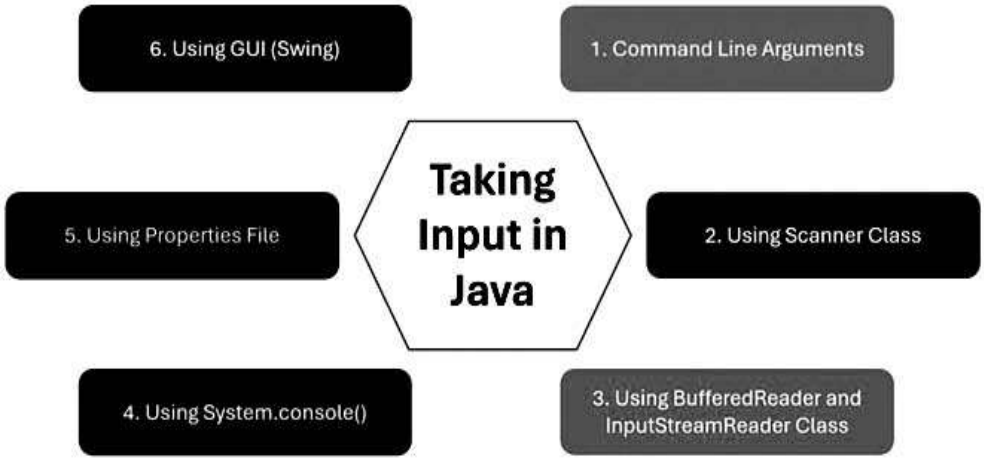


Fig. 6.4 : Converting Arguments to Other Data Types

Examples :

```
int x = Integer.parseInt(args[0]);
double d = Double.parseDouble(args[1]);
```

Advantages of Command Line Arguments

- No need for recompilation
- Flexible input
- Suitable for automation
- Improves program usability

Limitations

- All arguments are strings
- No validation by default

NOTES

- Errors if arguments are missing

Real-World Applications

- File processing utilities
- Batch programs
- System tools
- Configuration-based applications

Activity 6.1.-Analyze Real-Life Use of Command Line Tools

Objectives:
To develop analytical thinking by connecting command line arguments with real-world business applications.

Materials Needed:
Internet access (optional), computer, notebook.

Activity Description:
Identify at least two real-world software tools or applications (such as data processing tools, automation scripts, or system utilities) that use command line arguments.
Analyze how inputs are passed and how they influence the output or behavior of the tool.
Document one example where command line usage could improve efficiency in a business task (e.g., bulk data processing, report generation).

Reflection/Outcome:
Summarize how command line arguments contribute to automation and efficiency in business environments, especially for repetitive or large-scale tasks.

☑ Check Your Progress 6.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What is type casting? Explain the difference between implicit type casting and explicit type casting.

. . . . . .

. . . . . .

. . . . . .

2. What are command line arguments? Explain how arguments are passed to a program.

. . . . . .

. . . . . .

. . . . . .

3. How can command line arguments be accessed in a program? Give an example.

. . . . . .

. . . . . .

. . . . . .

6.5. SUMMARY

This unit introduced important programming concepts essential for structured Java program development. The unit focused on enhancing logical thinking and program flow control. Concepts discussed in this unit help in decision-making and repetitive task execution. Proper use of these constructs improves program readability and maintainability. Learners gained insights into how Java programs behave during execution.

The unit emphasized correct syntax, logical accuracy, and disciplined coding practices. These concepts are widely used in real-world Java applications. Understanding this unit is essential for progressing to advanced Java programming topics.

The unit strengthens the learner's ability to design efficient and reliable programs. Mastery of these ideas forms a strong base for object-oriented programming concepts.

6.6. GLOSSARY

- **Program Flow –** The order in which statements are executed
- **Logic –** The reasoning behind program decisions
- **Condition –** An expression that evaluates to true or false
- **Execution –** The process of running a program
- **Syntax –** Rules that define correct program structure
- **Statement –** An instruction in a program
- **Block –** A group of statements enclosed within braces
- **Iteration –** Repeated execution of statements
- **Decision Making –** Selecting actions based on conditions
- **Control Structure –** A construct that alters normal program flow

6.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** Type casting is the process of converting one data type into another.
   - Implicit Type Casting (Automatic):
     - Done automatically by the compiler.

NOTES

- Occurs when converting a smaller data type into a larger data type.
- Example:

```
inta=10;
floatb=a;  // int to float (automatic)
```

- Explicit Type Casting (Manual):
  - Done manually by the programmer using casting operators.
  - Used when converting a larger data type into a smaller one or when required.
  - Example:

```
floata=10.5;
intb= (int)a;  // float to int (manual)
```

2. **Answer :** Command line arguments are inputs given to a program at the time of execution through the command line.
   - These arguments are passed after the program name.
   - They allow users to provide input without modifying the program.

   Example (C) :

   program_name arg1 arg2 arg3

   Here, arg1, arg2, and arg3 are command line arguments.

3. **Answer :** Command line arguments are accessed using parameters of the main() function.

   In C:

   intmain(intargc, char*argv[])
   - argc (Argument Count): Number of arguments passed
   - argv (Argument Vector): Array of argument strings

6.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3747	

6.9. REFERENCES

- Balagurusamy, E. *Programming with Java*. McGraw-Hill.
- Schildt, H. *Java: The Complete Reference*. McGraw-Hill.
- Deitel, P., & Deitel, H. *Java How to Program*. Pearson.

- Oracle. *Java Documentation.*

6.10. EXERCISE

Part 1

A. Multiple Choice Questions :

1. The term program flow refers to:
   (a) Memory allocation process
   (b) Compilation speed
   (c) Order in which program statements are executed
   (d) Syntax rules only
2. Logical constructs in a program are mainly used to:
   (a) Store data
   (b) Improve hardware performance
   (c) Control decision making and execution
   (d) Increase file size
3. Which of the following improves readability of a program?
   (a) Random logic
   (b) Poor naming conventions
   (c) Structured and well-organized code
   (d) Ignoring indentation
4. A block of statements in Java is enclosed within:
   (a) ( )
   (b) [ ]
   (c) { }
   (d) < >
5. Correct program logic mainly ensures:
   (a) Faster execution only
   (b) More memory usage
   (c) Correct and expected output
   (d) Larger program size

Answers with Explanation :

1. Ans. (c)

Explanation : Program flow defines how statements are executed during runtime.

2. Ans. (c)

Explanation : Logical constructs guide program behavior based on conditions.

3. Ans. (c)

NOTES

Explanation : Explanation: Structured code is easier to read, debug, and maintain.

4. Ans. (c)

Explanation : Curly braces define a block of code.

5. Ans. (c)

Explanation : Correct logic leads to correct program results.

B. Short Answer Questions (50–100 words each)

1. Define program execution flow.
2. What is the importance of logical structure in programming?
3. Explain the term block of code.
4. Why is structured programming preferred in Java?
5. How does logical correctness affect program output?

Part 2

C. Long Answer Questions (200 words each)

1. Explain the importance of structured and logical program design in Java.
2. Describe how proper program flow improves reliability and maintainability of software.

Part 3

D. Descriptive Answer Questions (300 words each)

1. Discuss the role of logical thinking in developing effective Java programs.
2. Explain how the concepts covered in this unit help in solving real-world programming problems.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | BCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC | A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

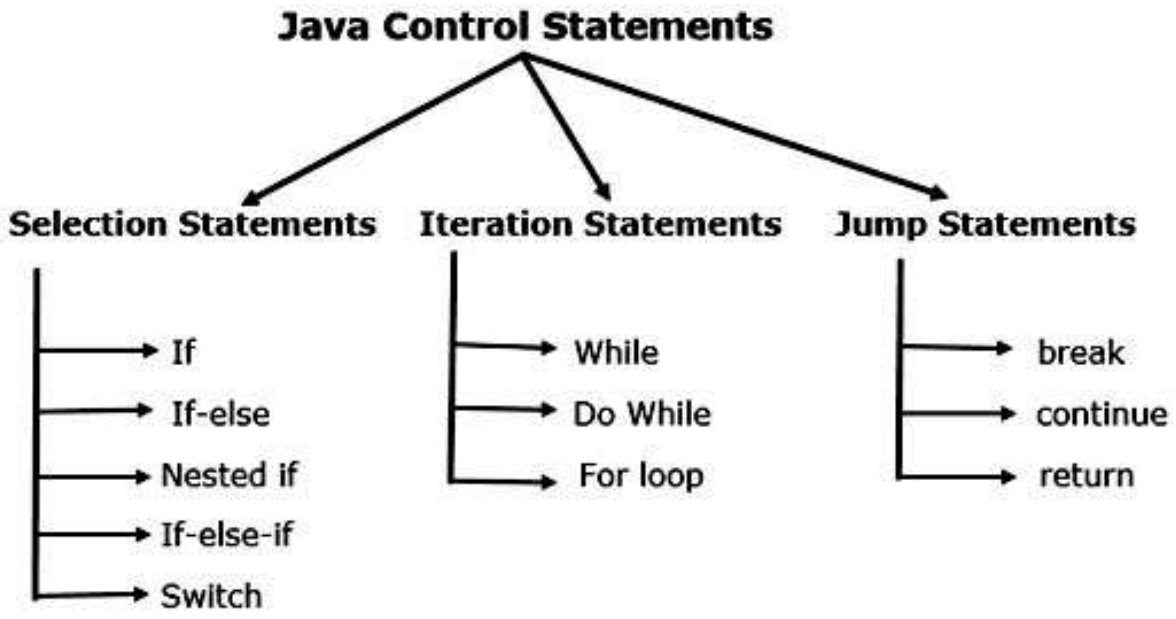
_____

_____

_____

***

Block III	OPERATORS AND DECISION CONTROL	95 – 128
Unit 9	Operators – I	97
Unit 10	Operators – II	107
Unit 11	Control Statements – Decision Making	117



This block focuses on the essential tools used to perform operations and control the flow of execution in Java programs. It provides a detailed study of different types of operators, including arithmetic, relational, logical, and assignment operators, across two units. Learners gain the ability to construct meaningful expressions and evaluate conditions effectively. By understanding how operators function, students develop the skills required to implement decision-making logic in programs. This block is crucial for building problem-solving abilities and forms a stepping stone toward more complex control structures in Java.

UNIT – 7
OPERATORS – I

UNIT STRUCTURE

7.1 Introduction
7.2 Objective
7.3 Arithmetic and Relational Operators
7.4 Logical and Assignment Operators
7.5 Summary
7.6 Glossary
7.7 Answer to check your progress
7.8 Suggested readings - OER
7.9 Reference
7.10 Exercise

7.1. INTRODUCTION

In the previous units, learners were gradually introduced to the fundamental and intermediate aspects of Java programming, including data handling, type conversion, and program execution techniques. Building on that progression, the present unit focuses on strengthening the learner's ability to apply Java programming concepts in a structured and logical manner. This unit is designed to deepen conceptual clarity and prepare learners for writing more organized, modular, and reliable programs. As programs increase in complexity, it becomes essential to use appropriate programming constructs and systematic approaches to solve problems efficiently. The concepts associated with this unit help learners move from basic understanding to practical application, enabling them to design solutions that are easier to read, debug, and maintain.

This unit emphasizes disciplined coding practices, logical thinking, and correct usage of programming mechanisms. It encourages learners to analyze problems carefully, choose suitable constructs, and implement solutions that follow standard programming conventions. These skills are crucial for both academic success and real-world software development. The knowledge gained in this unit serves as a foundation for advanced Java topics and object-oriented programming concepts. Mastery of these ideas will help learners develop confidence in program design and improve their ability to work on larger and more complex applications.

NOTES

7.2. OBJECTIVES

After completing this unit, the learner will be able to:

- Identify the key programming concepts introduced in this unit
- Explain the purpose and importance of these concepts in Java
- Apply the concepts to develop logically correct Java programs
- Analyze program behavior based on given conditions or structures
- Differentiate between various constructs covered in the unit
- Construct Java programs using appropriate programming techniques

7.3. ARITHMETIC AND RELATIONAL OPERATORS

Operators are special symbols in Java that perform specific operations on one or more operands. Arithmetic and relational operators are among the most frequently used operators in Java programming. They form the foundation for calculations, comparisons, decision-making, and control flow in programs. Arithmetic operators are used to perform mathematical calculations such as addition, subtraction, multiplication, and division. Relational operators are used to compare two values and determine the relationship between them. These comparisons usually result in a Boolean value (true or false), which is essential for conditional statements and logical expressions. Understanding arithmetic and relational operators is crucial for writing meaningful Java programs, as almost every real-world application involves calculations and comparisons.

7.3.1. Arithmetic Operators

Arithmetic operators in Java are used to perform basic mathematical operations on numeric data types such as int, float, double, and long. These operators work on operands and return a calculated result. Java supports both binary arithmetic operators (which require two operands) and unary arithmetic operators (which require a single operand).

List of Arithmetic Operators

Operators	Result
+	Addition of two numbers
-	Subtraction of two numbers
*	Multiplication of two numbers
/	Division of two numbers
%	(Modulus Operator)Divides two numbers and returns the remainder

Operator	Name	Description
+	Addition	Adds two operands
-	Subtraction	Subtracts one operand from another
*	Multiplication	Multiplies two operands
/	Division	Divides one operand by another
%	Modulus	Returns remainder after division
++	Increment	Increases value by 1
--	Decrement	Decreases value by 1

Examples of Arithmetic Operators

```
int a = 10;
int b = 3;

System.out.println(a + b);  // 13
System.out.println(a - b);  // 7
System.out.println(a * b);  // 30
System.out.println(a / b);  // 3
System.out.println(a % b);  // 1
```

Unary Arithmetic Operators

Unary operators act on a single operand.

```
int x = 5;
x++;   // x becomes 6
x--;   // x becomes 5
```

Pre-Increment and Post-Increment

Increment/decrement Operators

Example	Name	Effect
++$a	Pre-increment	Increments *$a* by one, then returns *$a*.
$a++	Post-increment	Returns *$a*, then increments *$a* by one.
--$a	Pre-decrement	Decrements *$a* by one, then returns *$a*.
$a--	Post-decrement	Returns *$a*, then decrements *$a* by one.

```
int x = 5;
System.out.println(++x);  // 6 (pre-increment)
System.out.println(x++);   // 6 (post-increment, then x
becomes 7)
```

Integer vs Floating-Point Division

```
int a = 10, b = 4;
System.out.println(a / b);      // 2

double c = 10;
System.out.println(c / b);      // 2.5
```

NOTES

This shows how data types affect arithmetic results.

Importance of Arithmetic Operators

- Used in calculations and formulas
- Essential for loops and counters
- Required for business logic and data processing
- Forms the base for scientific and financial applications

7.3.2. Relational Operators

Relational operators are used to compare two operands and determine the relationship between them. The result of a relational operation is always a Boolean value (true or false). These operators are commonly used in decision-making statements such as if, while, and for.

List of Relational Operators

Java Symbol	Operator
==	Equal To (See Caution Below)
!=	Not Equal To
<	Less Than
<=	Less Than Or Equal To
>	Greater Than
>=	Greater Than or Equal To
&&	AND
||	OR

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Examples of Relational Operators

```
int x = 10;
int y = 20;

System.out.println(x == y);   // false
System.out.println(x != y);   // true
System.out.println(x > y);    // false
System.out.println(x < y);    // true
```

Relational Operators in Decision Making

```
int marks = 65;
if (marks >= 40) {
    System.out.println("Pass");
}
```

Common Mistakes

```
if (x = y)   // incorrect
if (x == y)  // correct
Using = instead of == is a common beginner error.
```

Importance of Relational Operators

- Used for comparisons
- Essential for conditional execution
- Helps in decision-making logic
- Supports loops and validations

7.4. LOGICAL AND ASSIGNMENT OPERATORS

Logical and assignment operators further enhance Java's ability to control program execution and manage variable values efficiently.

7.4.1. Logical Operators

Concept of Logical Operators

Logical operators are used to combine multiple conditions and evaluate complex expressions. They operate on boolean values and return a boolean result.

List of Logical Operators

Truth table
for
Logical operators

A	B	A|B	A&B	A^B	~A
false	false	false	false	false	true
false	true	true	false	true	true
true	false	true	false	true	false
true	true	true	true	false	false

Operator	Name	Description
&&	Logical AND	True if both conditions are true
`		`
!	Logical NOT	Reverses the condition

Examples of Logical Operators

```
int a = 10, b = 5;

System.out.println(a > b && b > 0);  // true
System.out.println(a < b || b > 0);  // true
System.out.println(!(a > b));        // false
```

NOTES

```
Short-Circuit Evaluation
if (a != 0 && b / a > 1) {
    System.out.println("Valid");
}
```

Java stops evaluation as soon as the result is known, improving efficiency.

Importance of Logical Operators

- Combine multiple conditions
- Control complex decisions
- Improve program flexibility
- Used in validations and loops

7.4.2. Assignment Operators

Assignment operators are used to assign values to variables. Java provides simple as well as compound assignment operators that simplify coding and improve readability.

Simple Assignment Operator

```
int x = 10;
```

Compound Assignment Operators

Operator	Example Usage	Equivalent Expression
=	a=b	a=b
+=	a+=b	a=a+b
.=	a.=b	a=ab
*=	a*=b	a=a*b
/=	a*=b	a=a/b
%=	a*=b	a=a%b

Operator	Example	Equivalent Expression
+=	*x* += 5	*x* = *x* + 5
-=	*x* -= 5	*x* = *x* - 5
*=	*x* *= 5	*x* = *x* * 5
/=	*x* /= 5	*x* = *x* / 5
%=	*x* %= 5	*x* = *x* % 5

Examples of Assignment Operators

```
int x = 10;
x += 5;   // x = 15
x *= 2;   // x = 30
```

Advantages of Assignment Operators

- Reduces code length
- Improves readability

- Minimizes typing errors
- Enhances performance slightly

Real-World Use Cases

- Counters and accumulators
- Loop control
- Updating values dynamically
- Mathematical computations

Activity 7.1 :

Use arithmetic and relational operators in a Java program to perform basic calculations. Display the results clearly.

☑ Check Your Progress 7.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What are arithmetic operators? Explain with examples.

. . . . . .

. . . . . .

. . . . . .

2. What are relational operators? List any four and explain their use.

. . . . . .

. . . . . .

. . . . . .

3. Explain logical operators and assignment operators with suitable examples.

. . . . . .

. . . . . .

. . . . . .

7.5. SUMMARY

This unit develops a strong foundation in structured programming through a focused study of operators in Java. It enhances the learner's ability to understand how different operators control program logic, enabling smoother execution flow and better organization of code. By applying these concepts effectively, learners are able to write programs that are clear, efficient, and easy to maintain. Emphasis is placed on disciplined coding practices and analytical thinking, guiding learners toward minimizing errors and improving overall program performance. The unit also highlights the practical relevance of operators, showing how they are widely used in real-world Java applications. Serving as a

NOTES

bridge between basic syntax and more advanced programming topics, this unit prepares learners for higher-level concepts. Overall, the knowledge and skills gained here contribute to building modular, scalable programs while strengthening both theoretical understanding and practical coding abilities.

7.6. GLOSSARY

- **Program Structure –** The organized arrangement of statements in a program
- **Logic –** The reasoning that controls program behavior
- **Execution Flow –** The order in which program statements run
- **Block –** A group of statements enclosed within braces
- **Condition –** An expression that evaluates to true or false
- **Iteration –** Repeated execution of statements
- **Statement –** A single instruction in a program
- **Syntax –** Rules that define correct program structure
- **Modularity –** Dividing a program into manageable parts
- **Maintainability –** Ease of updating and modifying a program

7.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer : Arithmetic operators** are used to perform mathematical operations on variables and values.

   Common arithmetic operators:

   - + (Addition)(Subtraction)
   - (Multiplication)
   - / (Division)
   - % (Modulus)

2. **Answer : Relational operators** are used to compare two values and return a Boolean result (true or false).

   Common relational operators:

   - == (Equal to)
   - != (Not equal to)
   - > (Greater than)
   - < (Less than)

3. **Answer : Logical operators** are used to combine multiple conditions:

   - && (AND)
   - || (OR)
   - ! (NOT)

Assignment operators are used to assign values to variables:

- = (Simple assignment)
- +=, -=, *=, /= (Compound assignment)

7.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3748	

7.9. REFERENCES

1. Balagurusamy, E. *Programming with Java*. McGraw-Hill.
2. Schildt, H. *Java: The Complete Reference*. McGraw-Hill.
3. Deitel, P., & Deitel, H. *Java How to Program*. Pearson.
4. Oracle. *Java Documentation*.

7.10. EXERCISE

Part 1

A. Multiple Choice Questions :

1. Which arithmetic operator is used to find the remainder of a division operation?

   (a) / (b) *
   (c) % (d) +

2. What is the result of the expression 10 / 4 when both operands are integers?

   (a) 2.5 (b) 2
   (c) 2.0 (d) 3

3. Which relational operator checks whether two values are equal?

   (a) = (b) !=
   (c) == (d) <=

4. Which logical operator returns true only if both conditions are true?

   (a) || (b) !
   (c) & (d) &&

5. Which assignment operator is equivalent to x = x + 5?

   (a) x =+ 5 (b) x += 5
   (c) x == 5 (d) x := 5

NOTES

Answers with Explanation :

1. Ans. (c)

Explanation: The modulus (%) operator returns the remainder after division.

2. Ans. (b)

Explanation : Integer division truncates the decimal part.

3. Ans. (c)

Explanation : == compares two values for equality.

4. Ans. (d)

Explanation : Logical AND (&&) requires both conditions to be true.

5. Ans. (b)

Explanation : += is a compound assignment operator.

B. Short Answer Questions (50–100 words each)

1. Explain arithmetic operators in Java with one example.
2. What are relational operators? Why are they used in decision making?
3. Define logical operators and list their types.
4. What is the difference between = and == operators?
5. Explain compound assignment operators with an example.

Part 2

C. Long Answer Questions (200 words each)

1. Explain all arithmetic operators in Java with suitable examples.
2. Describe relational and logical operators and explain how they are used in conditional statements.

Part 3

D. Descriptive Answer Questions (300 words each)

1. Discuss arithmetic, relational, and logical operators in Java. Explain their role in program control and decision making with examples.
2. Explain assignment operators in Java. Compare simple assignment and compound assignment operators with suitable examples.

UNIT – 8
OPERATORS – II

UNIT STRUCTURE

8.1 Introduction
8.2 Objective
8.3 Bitwise and Conditional Operators
8.4 Increment, Decrement and Expressions
8.5 Summary
8.6 Glossary
8.7 Answer to check your progress
8.8 Suggested readings - OER
8.9 Reference
8.10 Exercise

8.1. INTRODUCTION

In the earlier units, learners gradually progressed from understanding basic Java syntax to applying operators and logical constructs for building meaningful programs. With this foundation in place, Unit 8 aims to further strengthen the learner's ability to design, analyze, and implement Java programs in a structured and systematic manner. This unit focuses on enhancing conceptual clarity and encouraging disciplined programming practices. As Java programs grow more complex, it becomes necessary to apply appropriate programming techniques that improve readability, correctness, and efficiency. The concepts associated with this unit help learners organize program logic effectively and manage execution in a controlled way. These ideas are essential for developing programs that are easy to understand, debug, and maintain.

This unit emphasizes analytical thinking and correct application of programming constructs. Learners are encouraged to understand why certain techniques are used and how they affect program behavior. Such understanding is crucial not only for academic assessments but also for real-world software development scenarios. The knowledge gained in this unit serves as a bridge between intermediate Java concepts and advanced application development. Mastery of these ideas will enable learners to approach problem-solving with confidence and prepare them for more advanced topics in Java programming.

8.2. OBJECTIVES

After completing this unit, the learner will be able to:

NOTES

- Identify the key programming concepts covered in this unit
- Explain the purpose and significance of these concepts
- Apply the concepts to develop correct and efficient Java programs
- Analyze program behavior based on given logic or structures
- Differentiate between various programming techniques discussed
- Construct Java programs using appropriate methods and constructs

8.3. BITWISE AND CONDITIONAL OPERATORS

Operators in Java are symbols that perform operations on variables and values. While arithmetic and logical operators deal with numeric calculations and Boolean logic, bitwise and conditional operators operate at a deeper level of computation and decision-making. These operators are especially useful in low-level programming, optimization tasks, flag manipulation, and compact decision expressions.

8.3.1. Bitwise Operators

Bitwise operators perform operations on individual bits of integer data types such as byte, short, int, and long. Instead of working on entire numeric values, these operators examine and manipulate each bit (0 or 1) within the binary representation of numbers. Understanding bitwise operators requires basic knowledge of binary number systems, where:

- 0 represents OFF
- 1 represents ON

Bitwise operators are commonly used in:

- System-level programming
- Embedded systems
- Cryptography
- Performance optimization
- Flag and mask handling

Binary Representation of Numbers

1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

9	1001
10	1010

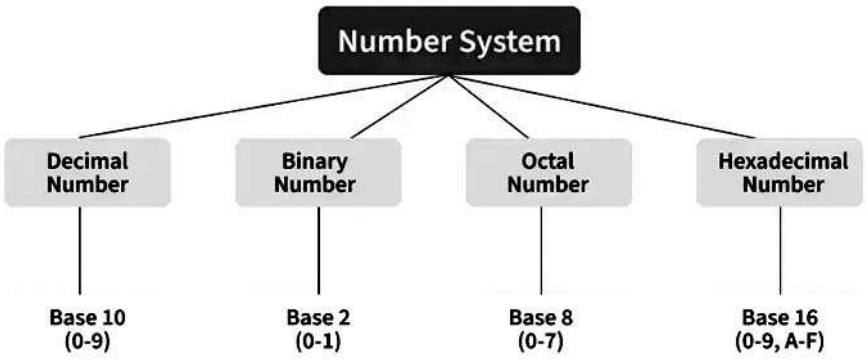


Fig. 8.1 : Binary Representation of Numbers

Example:

Decimal 5 → Binary 0101

Decimal 3 → Binary 0011

Types of Bitwise Operators in Java

Operator	Name	Description
&	Bitwise AND	1 if both bits are 1
`	`	Bitwise OR
^	Bitwise XOR	1 if bits are different
~	Bitwise Complement	Inverts all bits
<<	Left Shift	Shifts bits left
>>	Right Shift	Shifts bits right
>>>	Unsigned Right Shift	Shifts bits right with zero fill

Bitwise AND (&)

The bitwise AND operator compares each bit of two operands.

Example

```
int a = 5;    // 0101
int b = 3;    // 0011
int result = a & b;  // 0001 → 1
```

Explanation :

Only bits that are 1 in both operands remain 1.

Bitwise OR (|)

The bitwise OR operator sets a bit to 1 if at least one operand bit is 1.

```
int result = a | b;  // 0111 → 7
```

Bitwise XOR (^)

The XOR operator sets a bit to 1 if the bits are different.

NOTES

```
int result = a ^ b;  // 0110 → 6
```

Bitwise Complement (~)

The complement operator inverts all bits.

```
int x = 5;
int y = ~x;
```

This converts 0101 to 1010 (two's complement form).

Shift Operators

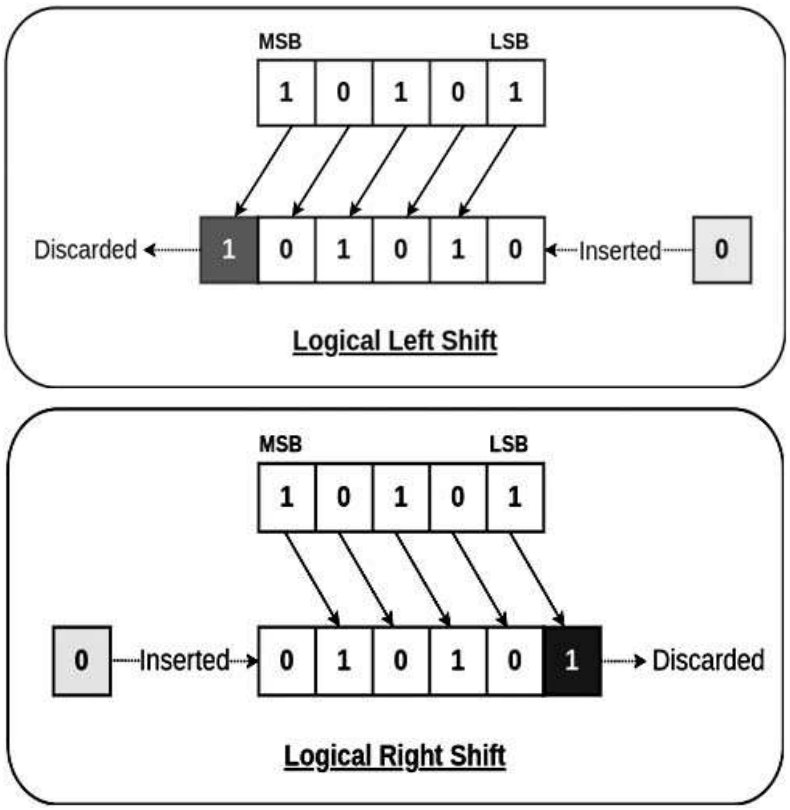


Fig. 8.2. Shift Operators.

Left Shift (<<)

```
int x = 5;       // 0101
int y = x << 1; // 1010 → 10
```

Right Shift (>>)

```
int y = x >> 1; // 0010 → 2
Unsigned Right Shift (>>>)
int z = x >>> 1;
```

Practical Applications of Bitwise Operators

- Setting and clearing flags
- Fast multiplication/division by powers of two
- Encoding multiple boolean values
- Low-level hardware communication

8.3.2. Conditional Operator

The conditional operator (?:) is the only ternary operator in Java. It provides a compact way to write conditional expressions and is often used as an alternative to simple if–else statements.

Syntax

```
condition ? expression1 : expression2;
```

- If condition is true → expression1 executes
- If condition is false → expression2 executes

Example

```
int a = 10, b = 20;
int max = (a > b) ? a : b;
```

Nested Conditional Operator

```
int max = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
```

☑ Check Your Progress 8.1.

Notes: (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What are bitwise operators? Explain any two with examples.

   . . . . . .
   . . . . . .
   . . . . . .

2. What is the conditional (ternary) operator? Explain its syntax with an example.

   . . . . . .
   . . . . . .
   . . . . . .

3. Differentiate between pre-increment and post-increment operators. Also explain expression evaluation with a simple example.

   . . . . . .
   . . . . . .
   . . . . . .

Advantages

- Reduces code length
- Improves readability for simple decisions
- Useful in assignments and expressions

Limitations

- Reduces clarity if overused
- Difficult to debug when nested

8.4. INCREMENT, DECREMENT AND EXPRESSIONS

Increment, decrement, and expressions are essential components of Java programming. They control how values change during execution and determine how complex expressions are evaluated.

8.4.1. Increment and Decrement Operators

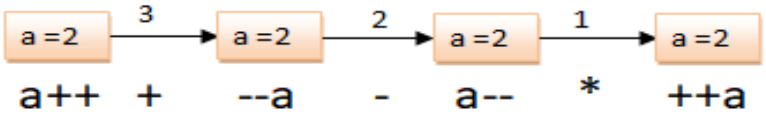
Concept

Increment (++) and decrement (--) operators are unary operators used to increase or decrease the value of a variable by one.

Types

- **Pre-increment (++x)**
- **Post-increment (x++)**
- **Pre-decrement (--x)**
- **Post-decrement (x--)**
- **Pre vs Post Increment**

Pre vs Post Increment



Examples in Loops

```
for(int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

Common Mistakes

```
int a = 5;
int b = a++ + ++a;  // confusing result
```

Best Practices

- Avoid complex expressions
- Use increment operators clearly
- Prefer simple statements

Activity 8.1. Business Decision Comparison Using Relational Logic

Objectives:
To understand and apply relational operators in evaluating business conditions and decision-making.
Materials Needed:
Notebook or spreadsheet tool

Activity Description:
Consider two business scenarios (e.g., two product investments or

marketing campaigns).

- Define at least 3 parameters such as cost, expected profit, and risk level (assign numerical values).
- Use relational operators (>, <, ==, !=, ≥, ≤) to compare the scenarios (e.g., Profit A > Profit B).
- Based on comparisons, decide which option is better and justify your choice logically.

Reflection/Outcome:
Briefly explain how relational comparisons help in making data-driven business decisions.

8.4.2. Expression Evaluation

An expression is a combination of variables, constants, operators, and method calls that evaluates to a value. Java evaluates expressions based on operator precedence and associativity.

Operator Precedence

Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ~ ! (type)	Right to left	Unary prefix
* / %	Left to right	Multiplicative
+ -	Left to right	Additive
<< >> >>>	Left to right	Shift
< <= > >=	Left to right	Relational
== !=	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

Example

```
int result = 10 + 5 * 2; // 20
```

Associativity

- Left-to-right for most operators
- Right-to-left for assignment and ternary

Use of Parentheses

```
int result = (10 + 5) * 2; // 30
```

Type Conversion in Expressions

```
int a = 10;
double b = 3;
double result = a / b;
```

NOTES

Importance of Expression Evaluation

- Prevents logical errors
- Ensures correct output
- Improves code reliability

8.5 SUMMARY

This unit focused on strengthening structured and logical Java programming skills. The concepts discussed help improve program organization and clarity. Proper application of these ideas leads to more reliable and maintainable code.

The unit emphasized disciplined coding practices and analytical thinking. Learners gained insights into managing program logic effectively. These concepts are widely applicable in real-world Java applications. The unit builds on earlier topics and prepares learners for advanced concepts. Understanding this unit helps reduce logical and runtime errors. The skills acquired support scalable and modular program design. Overall, the unit enhances both theoretical understanding and practical programming ability

8.6. GLOSSARY

- **Program Logic –** The sequence of reasoning that controls program execution
- **Execution Flow –** The order in which program statements are executed
- **Structure –** Organized arrangement of program components
- **Block –** A group of statements enclosed within braces
- **Condition –** An expression that evaluates to true or false
- **Iteration –** Repeated execution of statements
- **Statement –** A single instruction in a program
- **Syntax –** Rules defining valid program structure
- **Modularity –** Dividing a program into manageable units
- **Maintainability –** Ease of modifying and updating a program

8.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** Bitwise operators **operate on individual bits of binary numbers**.

   Common bitwise operators include:

   - & (AND)
   - | (OR)
   - ^ (XOR)
   - ~ (NOT)

2. Answer : The conditional operator (?:) **is used to make decisions in a single line. It is also called the ternary operator**.

Syntax :

condition?expression1 : expression2;

3. Answer : Pre-increment (++x) : Value is increased before use.

- **Post-increment (x++) :** Value is increased after use.

8.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3749	

8.9. REFERENCES

1. Balagurusamy, E. *Programming with Java*. McGraw-Hill.
2. Schildt, H. *Java: The Complete Reference*. McGraw-Hill.
3. Deitel, P., & Deitel, H. *Java How to Program*. Pearson.
4. Oracle. *Java Documentation*.

8.10 EXERCISE

Part 1

A. Multiple Choice Questions

1. The primary purpose of applying structured programming concepts is to:
   (a) Increase program length
   (b) Reduce compilation time
   (c) Improve program clarity and logical organization
   (d) Eliminate syntax rules
2. Proper organization of program logic mainly helps in:
   (a) Increasing hardware speed
   (b) Reducing memory size
   (c) Improving readability and maintainability
   (d) Avoiding variable declaration
3. A block of statements in Java is enclosed within:
   (a) ( ) (b) [ ] (c) { } (d) < >
4. Program execution flow refers to:
   (a) Storage of variables
   (b) Input mechanism
   (c) Order in which statements are executed
   (d) Output formatting

NOTES

5. Logical correctness in a program ensures:
   (a) Faster compilation
   (b) More memory usage
   (c) Correct and expected output
   (d) Larger program size

Answers with Explanation :

1. Ans. (c)

Explanation : Structured programming helps organize code logically, making it easier to read and maintain.

2. Ans. (c)

Explanation : Well-organized logic makes programs easier to understand and modify.

3. Ans. (c)

Explanation: Curly braces group multiple statements into a single block.

4. Ans. (c)

Explanation : Execution flow determines how instructions run during program execution.

5. Ans. (c)

Explanation : Correct logic leads to correct program behavior and output.

B. Short Answer Questions (50–100 words each)

1. Explain the importance of structured programming in Java.
2. What is meant by program execution flow?
3. Define a block of code and state its importance.
4. Why is logical correctness essential in program development?
5. How does modular programming improve software quality?

Part 2

C. Long Answer Questions (200 words each)

1. Explain how proper program structure improves readability and maintainability of Java programs.
2. Describe the role of logical organization in developing reliable software applications.

Part 3

D. Descriptive Answer Questions (300 words each)

1. Discuss the importance of logical thinking and structured design in Java programming.
2. Explain how the concepts covered in this unit contribute to efficient and maintainable software development.

UNIT – 9
CONTROL STATEMENTS – DECISION MAKING

UNIT STRUCTURE

9.1 Introduction
9.2 Objective
9.3 Conditional Statements
9.4 Case Control Statements
9.5 Summary
9.6 Glossary
9.7 Answer to check your progress
9.8 Suggested readings - OER
9.9 Reference
9.10 Exercise

9.1. INTRODUCTION

In the previous units, learners explored various programming constructs and techniques that form the foundation of Java application development. As students' progress further, it becomes increasingly important to integrate these concepts in a structured and systematic manner. Unit 9 is designed to strengthen the learner's understanding of advanced program organization and logical implementation techniques. This unit focuses on developing analytical thinking and disciplined coding practices that help learners handle more complex programming requirements. The concepts related to this unit encourage students to move beyond isolated statements and begin working with well-structured program logic. Such skills are essential for building programs that are not only correct but also efficient and easy to maintain.

The topics associated with this unit are commonly used in real-world software development, where programs must handle different conditions, manage execution flow effectively, and produce reliable output. By mastering the ideas presented in this unit, learners will be better equipped to design programs that are scalable, modular, and robust. Overall, this unit acts as a bridge between intermediate Java concepts and more advanced application development topics. The

NOTES

knowledge gained here will support learners in academic assessments as well as in practical programming scenarios.

9.2. OBJECTIVES

After completing this unit, the learner will be able to:

- Identify the key programming concepts introduced in this unit
- Explain the significance of these concepts in Java programming
- Apply the concepts to develop correct and efficient Java programs
- Analyze program behavior under different conditions
- Differentiate between related programming constructs
- Construct Java programs using appropriate logic and structure

9.3. CONDITIONAL STATEMENTS

Conditional statements enable a Java program to make decisions based on conditions. Instead of executing instructions sequentially, the program evaluates expressions and chooses different execution paths. This capability is essential for validations, user interactions, business rules, and dynamic logic.

Conditional statements in Java are primarily:

- if
- if–else
- nested if

These constructs rely on Boolean expressions (true/false), typically formed using relational and logical operators.

9.3.1. if Statement

The if statement is the simplest decision-making construct in Java. It allows execution of a block of code only when a specified condition evaluates to true. If the condition is false, the block is skipped.

Syntax

```
if (condition) {
    // statements
}
```

How It Works

1. The condition is evaluated.
2. If the condition is true → the block executes.
3. If false → the block is ignored and control moves to the next statement.

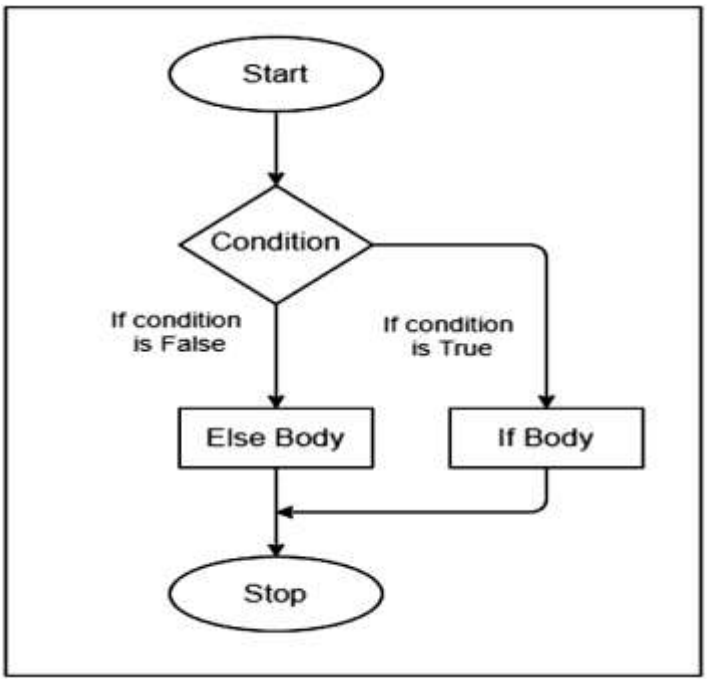


Fig. 9.1 :

Basic Example

```
int age = 20;
if (age >= 18) {
    System.out.println("Eligible to vote");
}
```

Real-World Examples

- Checking minimum balance in a bank account
- Validating login credentials
- Ensuring input values fall within limits

```
int balance = 5000;
if (balance < 1000) {
    System.out.println("Low balance warning");
}
```

Common Mistakes

```
if (age = 18)   // incorrect
if (age == 18) // correct
```

Best Practices

- Keep conditions simple and readable
- Use meaningful variable names
- Avoid overly complex expressions

9.3.2. if–else Statement

The if–else statement extends the if statement by providing an alternative execution path when the condition is false. It ensures that exactly one block executes.

NOTES

Syntax

```
if (condition) {
    // true block
} else {
    // false block
}
```

Execution Flow

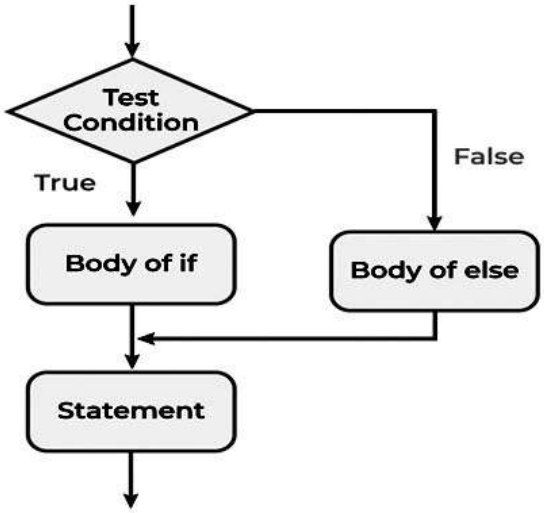


Fig. 9.2 : Execution Flow.

1. Condition is evaluated.
2. If true → if block executes.
3. If false → else block executes.

Example

```
int marks = 35;
if (marks >= 40) {
    System.out.println("Pass");
} else {
    System.out.println("Fail");
}
```

Multiple if–else Ladder

```
if (marks >= 75) {
    System.out.println("Distinction");
} else if (marks >= 60) {
    System.out.println("First Class");
} else if (marks >= 40) {
    System.out.println("Pass");
} else {
    System.out.println("Fail");
}
```

Use Cases

- Grade calculation
- Salary slab computation
- Menu-based programs

Advantages

- Clear decision branching
- Better readability than nested if for linear conditions

9.3.3. Nested if

A nested if is an if statement placed inside another if or else block. It is used when decisions depend on multiple hierarchical conditions.

Syntax

```
if (condition1) {
    if (condition2) {
        // statements
    }
}
```

Execution Logic

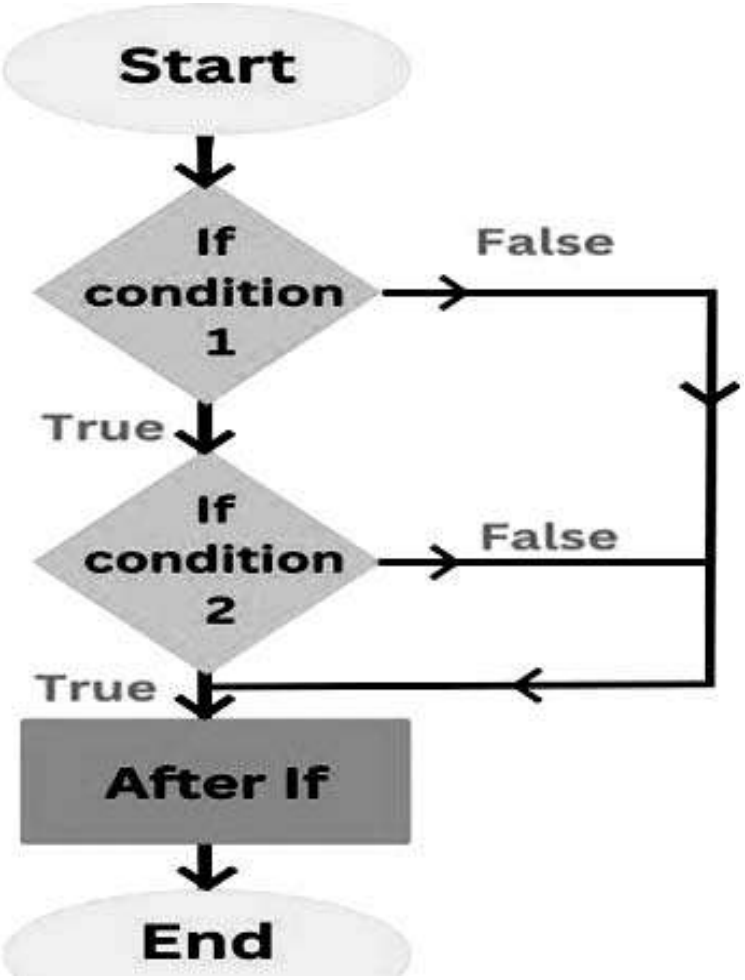


Fig. 9.3 (a) : Execution Logic.

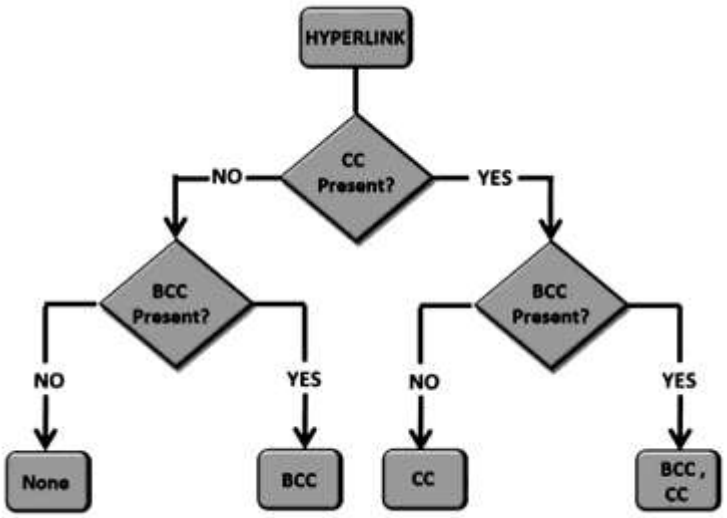


Fig. 9.3 (b) : Execution Logic.

Example

```
int age = 22;
boolean hasID = true;

if (age >= 18) {
    if (hasID) {
        System.out.println("Entry allowed");
    }
}
```

Complex Nested Example

```
int a = 10, b = 20, c = 15;
if (a > b) {
    if (a > c) {
        System.out.println("a is largest");
    }
} else {
    if (b > c) {
        System.out.println("b is largest");
    }
}
```

Drawbacks

- Reduces readability if deeply nested
- Harder to debug and maintain

Best Practices

- Limit nesting depth
- Consider else-if ladder where appropriate
- Use proper indentation

9.4. CASE CONTROL STATEMENTS

NOTES

Case control statements allow a program to select one execution path from many options based on a single variable's value. Java provides the switch statement for this purpose.

9.4.1. switch Statement

The switch statement evaluates an expression and matches it against multiple case labels. When a match is found, the corresponding block executes.

Syntax

```
switch (expression) {
    case value1:
        // statements
        break;
    case value2:
        // statements
        break;
    default:
        // statements
}
```

Supported Types

- byte, short, int, char
- String (Java 7 onwards)
- enum

Execution Flow

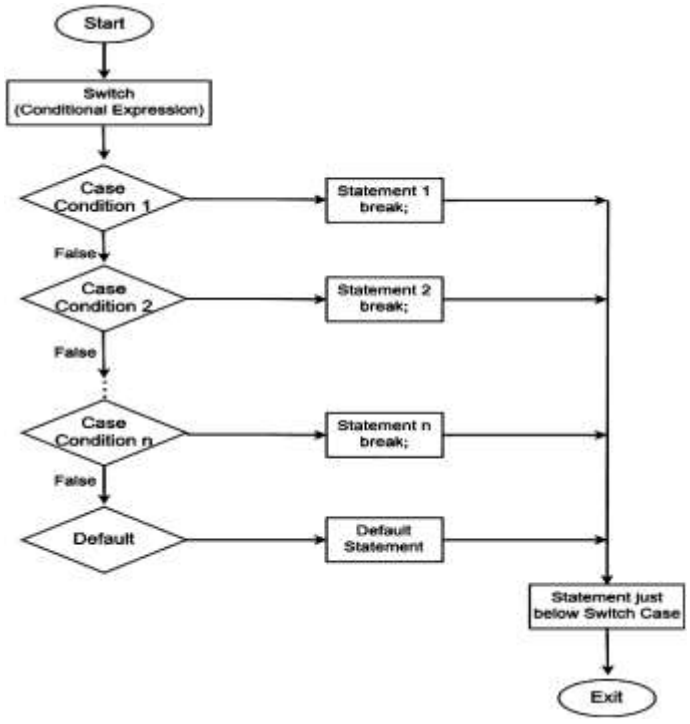


Fig. 9.4 : Execution Flow

NOTES

Example

```
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

Advantages Over if–else

- Cleaner code for multiple fixed values
- Better readability
- Faster execution in many cases

Limitations

- Cannot use relational conditions
- Requires constant case values

9.4.2. break Usage

The break statement terminates the current switch case and transfers control outside the switch block. Without break, execution continues into the next case—this is called fall-through.

Example Without break

```
int x = 1;
switch (x) {
    case 1:
        System.out.println("One");
    case 2:
        System.out.println("Two");
}
```

Output

One

Two

Example With break

```
switch (x) {
    case 1:
```

```
            System.out.println("One");
            break;
        case 2:
            System.out.println("Two");
            break;
    }
```

Intentional Fall-Through

```
    switch (month) {
        case 12:
        case 1:
        case 2:
            System.out.println("Winter");
            break;
    }
```

Activity 9.1.

Create a program using if-else or switch to solve a real-life problem (e.g., grading system). Test with different inputs.

☑ Check Your Progress 9.1.

1. What are conditional statements in Java? Explain the if, if–else, and nested if statements.

   . . . . . .
   . . . . . .
   . . . . . .

2. What is a switch statement? How does it work in Java?

   . . . . . .
   . . . . . .
   . . . . . .

3. What is the use of the break statement in switch-case?

   . . . . . .
   . . . . . .
   . . . . . .

9.5. SUMMARY

This unit on Control Statements – Decision Making is designed to strengthen structured and logical programming skills in Java. It emphasizes how decision-making constructs, such as conditional statements, help control the flow of a program, enabling developers to execute code based on specific conditions. Through this unit, learners develop a clearer understanding of how to organize programs

NOTES

effectively, improving both clarity and maintainability. The concepts encourage analytical thinking and disciplined coding practices, allowing programmers to manage logic more efficiently and reduce common logical and runtime errors. By applying these ideas, learners gain the ability to design modular and scalable programs that are reliable and adaptable to real-world applications. This unit also builds on previously learned programming fundamentals and lays a strong foundation for more advanced topics, ultimately enhancing both theoretical knowledge and practical coding proficiency.

9.6. GLOSSARY

- **Program Logic –** The sequence of reasoning that controls program execution
- **Execution Flow –** The order in which program statements are executed
- **Structure –** Organized arrangement of program components
- **Block –** A group of statements enclosed within braces
- **Condition –** An expression that evaluates to true or false
- **Iteration –** Repeated execution of statements
- **Statement –** A single instruction in a program
- **Syntax –** Rules defining valid program structure
- **Modularity –** Dividing a program into manageable units
- **Maintainability –** Ease of modifying and updating a program

9.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** Conditional statements in Java are used to make decisions based on conditions.
   - The **if statement** executes a block of code if a condition is true.
   - The **if–else statement** provides two choices: one block executes if the condition is true, otherwise another block executes.
   - **Nested if** means using an if statement inside another if, allowing multiple conditions to be checked.
2. **Answer :** The switch statement is used to select one of many code blocks to execute based on the value of a variable. It compares the variable with different case values and executes the matching case block. It improves readability when multiple conditions are involved.
3. **Answer :** The break statement is used to terminate the execution of a case in a switch statement. Without break, the program continues executing the next cases (fall-through), which may lead to unintended results.

9.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3750	

9.9. REFERENCES

- Balagurusamy, E. *Programming with Java*. McGraw-Hill.
- Schildt, H. *Java: The Complete Reference*. McGraw-Hill.
- Deitel, P., & Deitel, H. *Java How to Program*. Pearson.
- Oracle. *Java Documentation*.

9.10. EXERCISE

Part 1

A. Multiple Choice Questions

1. The main objective of applying structured programming concepts is to:
   (a) Increase program length
   (b) Reduce execution speed
   (c) Improve logical clarity and program organization
   (d) Eliminate syntax rules
2. Logical constructs in Java programs are mainly used to:
   (a) Store data permanently
   (b) Improve hardware performance
   (c) Control the execution flow of a program
   (d) Compress program files
3. A block of code in Java is enclosed within:
   (a) ( ) (b) [ ]
   (c) { } (d) < >
4. Program execution flow refers to:
   (a) Input method used
   (b) Output formatting style
   (c) Order in which program statements are executed
   (d) Memory allocation process
5. Logical correctness in a program ensures:
   (a) Faster compilation
   (b) More memory usage
   (c) Correct and expected output

NOTES

(d) Larger program size

Answers with Explanation :

1. Ans. (c)

Explanation : Structured programming improves readability, correctness, and maintainability.

2. Ans. (c)

Explanation : Logical constructs guide decisions and control how statements execute.

3. Ans. (c)

Explanation : Curly braces group multiple statements into a block.

4. Ans. (c)

Explanation : Execution flow determines how instructions run step by step.

5. Ans. (c)

Explanation : Correct logic leads to correct program behavior and results.

B. Short Answer Questions (50–100 words each)

1. What is meant by structured programming?
2. Explain the importance of program execution flow.
3. Define a block of code and explain its significance.
4. Why is logical correctness essential in program development?
5. How does modular program design improve software quality?

Part 2

C. Long Answer Questions (200 words each)

1. Explain how structured program design improves readability and maintainability of Java programs.
2. Describe the role of logical organization in developing reliable and efficient software.

Part 3

D. Descriptive Answer Questions (300 words each)

1. Discuss the importance of logical thinking and structured design in Java programming.
2. Explain how the concepts covered in this unit help in developing scalable and maintainable software systems.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | BCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC : A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

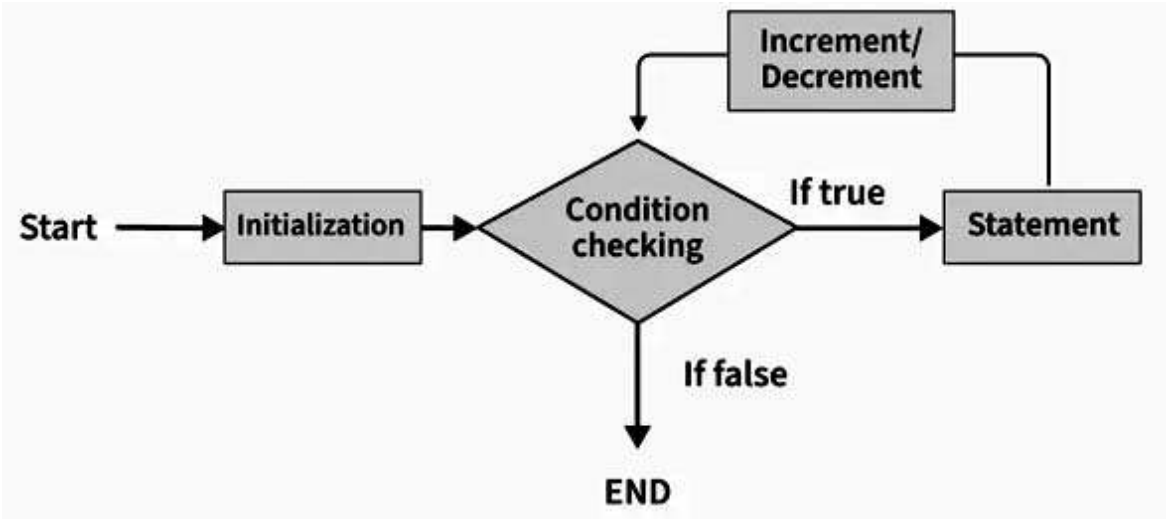
_____

_____

_____

***

Block IV	LOOPING STRUCTURES AND CORE DATA STRUCTURES	131 – 176
Unit 10	Control Statements – Looping	133
Unit 11	Arrays	151
Unit 12	String Handling	165



This block explores iterative control structures and fundamental data handling techniques in Java. It introduces looping constructs that allow repetitive execution of code, enhancing efficiency and reducing redundancy. The block also covers arrays, which enable the storage and manipulation of multiple data elements, and string handling, which is essential for text processing. Together, these topics help learners manage and organize data effectively while implementing dynamic program logic. By mastering these concepts, students gain the ability to design programs that handle large datasets and repetitive tasks with ease.

UNIT – 10
CONTROL STATEMENTS – LOOPING

UNIT STRUCTURE

10.1 Introduction
10.2 Objective
10.3 Iteration Statements
10.4 Jump and Labeled Statements
10.5 Summary
10.6 Glossary
10.7 Answer to check your progress
10.8 Suggested readings - OER
10.9 Reference
10.10 Exercise

10.1. INTRODUCTION

In the preceding units, learners have gradually progressed through the essential concepts of Java programming, beginning with basic syntax and operators and moving toward decision-making structures and control mechanisms. As students advance further, it becomes necessary to integrate previously learned concepts and apply them in a more structured, organized, and logical manner. Unit 10 is designed to strengthen this ability by emphasizing clarity of program structure and correctness of execution. This unit focuses on refining the learner's understanding of how Java programs are organized and how different programming constructs work together to solve problems efficiently. The concepts related to this unit encourage students to think systematically, analyze program logic carefully, and implement solutions that are easy to understand and maintain. Such skills are essential when dealing with larger programs or real-world software applications. The ideas covered in this unit are widely used in practical programming scenarios, where programs must handle varying inputs, manage execution flow effectively, and produce reliable outputs. By mastering these concepts, learners will be better equipped to write robust Java programs and avoid common logical and structural errors. Overall, this unit serves as an important step toward advanced Java programming and application development. It helps

NOTES

learners build confidence in program design and prepares them for more complex programming challenges.

10.2. OBJECTIVES

After completing this unit, the learner will be able to:

- Identify the key programming concepts introduced in this unit
- Explain the significance of these concepts in Java programming
- Apply the concepts to develop logically correct Java programs
- Analyze program behavior under different conditions
- Differentiate between various programming constructs discussed
- Construct Java programs using appropriate logic and structure

10.3. ITERATION STATEMENTS

Iteration statements, also known as looping statements, are used in Java to execute a block of code repeatedly as long as a specified condition remains true. Iteration is a fundamental concept in programming because many real-world problems involve repetition, such as processing a list of data, displaying records, calculating totals, or validating input. Without iteration statements, programmers would need to write the same code multiple times, which leads to redundancy, increased program size, and difficulty in maintenance. Java provides powerful and flexible iteration constructs that help developers write concise, efficient, and readable programs.

Java supports three primary iteration statements:

- **for loop**
- **while loop**
- **do–while loop**

Each loop differs in structure and usage, and the choice of loop depends on the programming requirement.

Need for Iteration Statements

Iteration statements are required in Java for the following reasons:

- To repeat tasks efficiently
- To reduce code duplication
- To process arrays and collections
- To perform calculations involving counters
- To build menu-driven and interactive programs

Classification of Iteration Statements

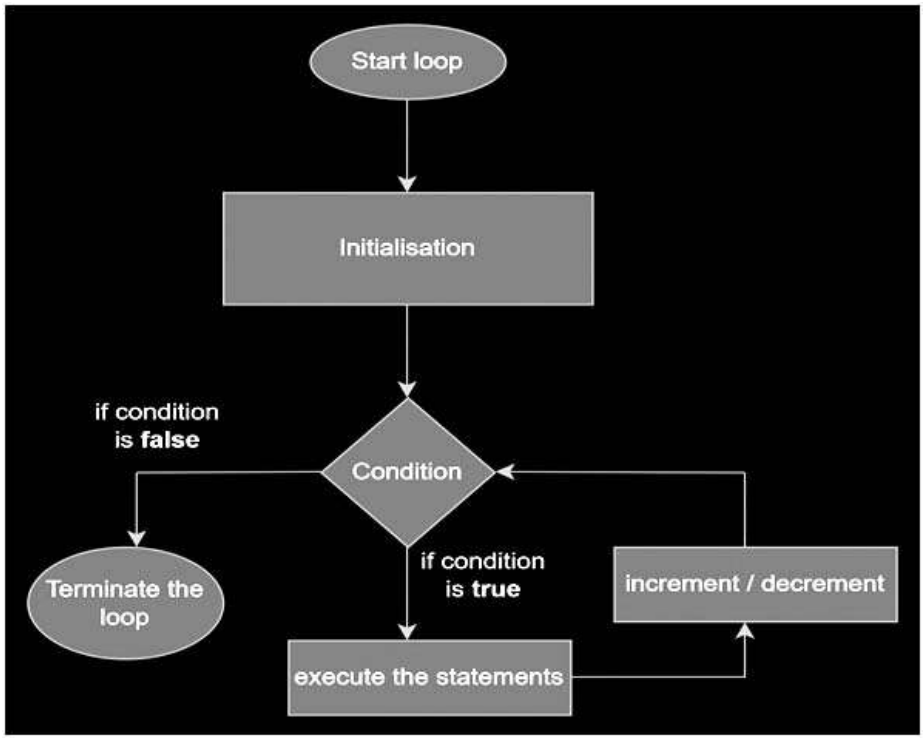


Fig. 10.1 : Classification of Iteration Statements.

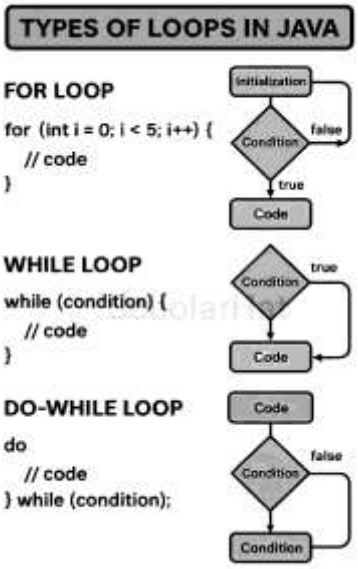


Fig. 10.2 : Types of Loops in Java.

Java iteration statements can be classified as:

1. Entry-controlled loops

- for loop
- while loop

2. Exit-controlled loop

- do–while loop

In entry-controlled loops, the condition is checked before executing the loop body, whereas in exit-controlled loops, the condition is checked after executing the loop body.

NOTES

10.3.1. for Loop

The for loop is an entry-controlled iteration statement that is used when the number of iterations is known in advance. It is compact and combines loop initialization, condition checking, and update in a single line.

Syntax

```
for (initialization; condition; increment/decrement) {
    // loop body
```

Execution Flow

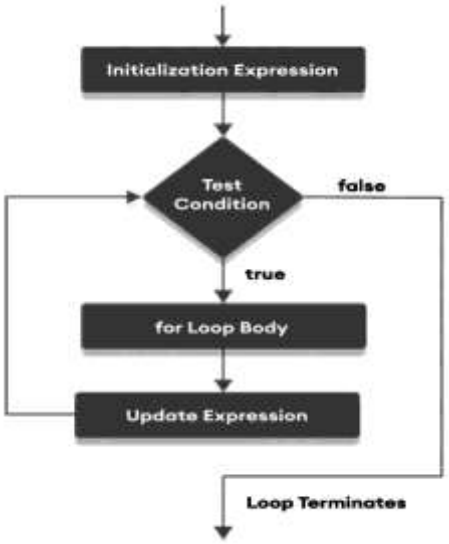


Fig. 10.3 : Execution flow



Fig. 10.4 :

1. Initialization is executed once
2. Condition is evaluated
3. If condition is true, loop body executes
4. Update statement executes
5. Control returns to condition check

Example

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

Output

1
2
3
4
5

Applications of for Loop

- Displaying number series
- Traversing arrays
- Generating tables
- Counting iterations

```
for (int i = 1; i <= 10; i++) {
    System.out.println("2 x " + i + " = " + (2 * i));
}
```

Nested for Loop

A loop inside another loop is called a **nested loop.**

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.print("* ");
    }
    System.out.println();
}
```

Common Errors in for Loop

- Missing increment/decrement
- Incorrect condition leading to infinite loop
- Using wrong loop boundary

Best Practices

- Use meaningful loop variables
- Avoid unnecessary nesting
- Clearly define loop limits

NOTES

10.3.2. while Loop

The while loop is also an entry-controlled loop, but it is used when the number of iterations is not known in advance. The loop continues executing as long as the condition remains true.

Syntax

```
while (condition) {
    // loop body
}
```

Execution Flow

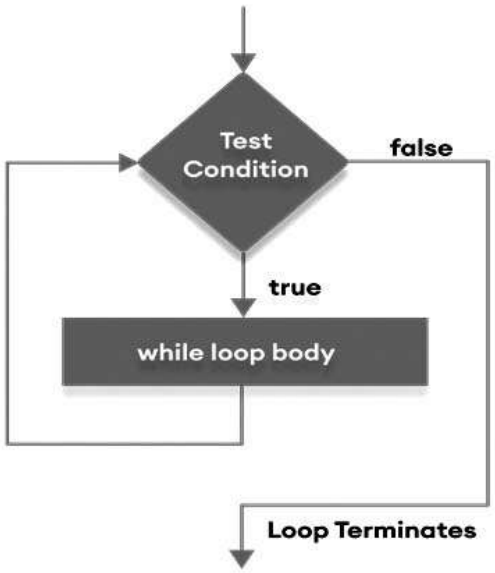


Fig. 10.5 (a) : Execution Flow.

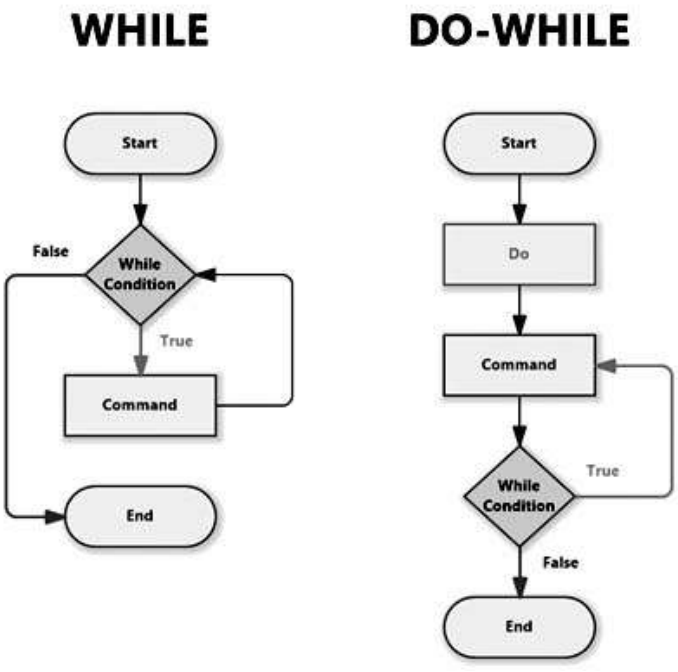


Fig. 10.5 (b) : Execution Flow.

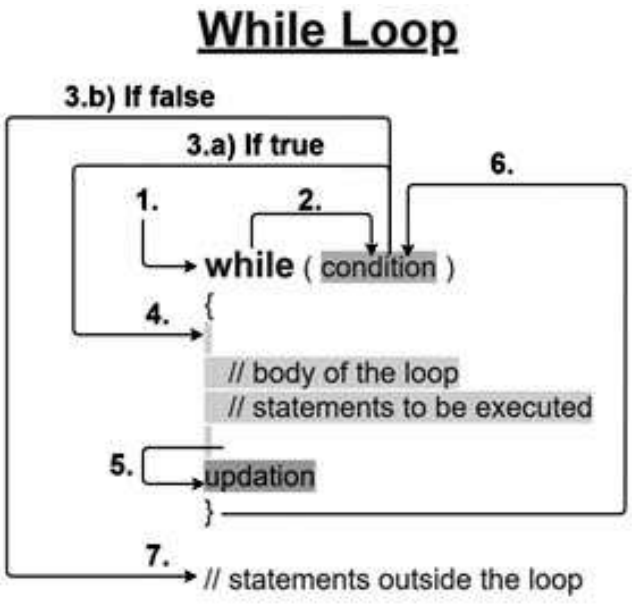


Fig. 10.5 (c) : Execution Flow.

Example

```
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;
}
```

Use Cases of while Loop

- Reading user input until a condition is met
- Menu-driven programs
- File and data stream processing

```
int choice = 1;
while (choice != 0) {
    System.out.println("Processing...");
    choice--;
}
```

Infinite while Loop

```
while (true) {
    System.out.println("Running...");
}
```

Infinite loops must be handled carefully to avoid program crashes.

Advantages

- Flexible condition control
- Suitable for uncertain iteration counts

Limitations

- Risk of infinite loops

NOTES

- Condition must be updated properly

10.3.3. do–while Loop

The do–while loop is an exit-controlled iteration statement. It guarantees that the loop body executes at least once, even if the condition is false.

Syntax

```
do {
    // loop body
} while (condition);
```

Execution Flow

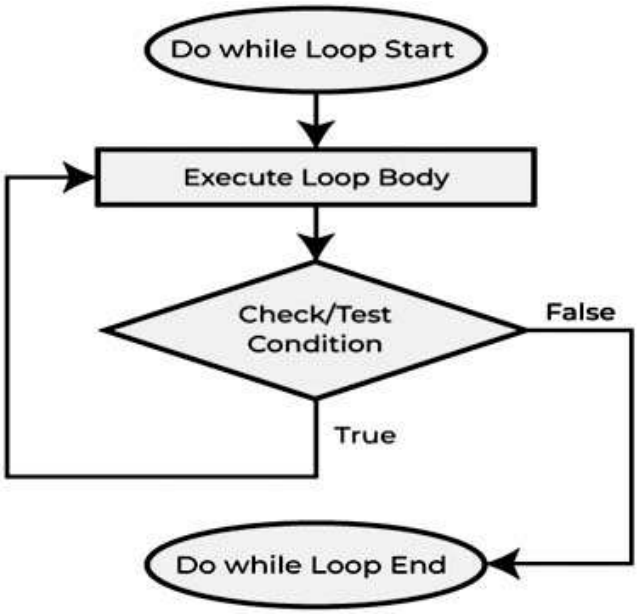


Fig. 10.6 : Execution Flow.

Example

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <= 5);
```

Practical Applications

- Menu-driven programs
- Input validation
- User interaction programs

Difference Between while and do–while

while loop	do–while loop
Entry-controlled	Exit-controlled
Condition checked first	Condition checked after execution
Loop may not execute	Executes at least once

Comparison of Iteration Statements

Feature	for	while	do–while
Condition check	Before	Before	After
Minimum execution	0 times	0 times	1 time
Best used when	Count known	Count unknown	At least one execution required

10.4. JUMP AND LABELED STATEMENTS

In Java programming, the normal execution of a program follows a sequential flow, where statements are executed one after another. However, in many practical situations, it becomes necessary to alter this normal flow based on certain conditions. Java provides jump statements to control the execution flow by allowing the program to exit a loop prematurely or skip specific iterations. In addition, labeled statements offer greater control when dealing with nested loops. Jump and labeled statements are especially useful in complex programs involving multiple loops, search operations, validation processes, and performance-critical logic. When used correctly, they improve program efficiency and readability. However, improper use can lead to confusing logic, so careful understanding is essential.

Java primarily provides the following jump and labeled control mechanisms:

- break statement
- continue statement
- Labeled loops

Purpose of Jump Statements

Jump statements are used in Java to:

- Exit a loop or switch statement early
- Skip the remaining part of the current loop iteration
- Control execution flow in nested loops
- Improve efficiency by avoiding unnecessary iterations

Without jump statements, programmers would need to use complex conditional logic, which can reduce clarity and increase error chances.

10.4.1. break and continue

The break statement is used to terminate the execution of a loop or switch statement immediately. When break is encountered, control is transferred outside the loop or switch block, and the remaining statements inside the loop are skipped.

Syntax

```
break;
```

Working of break Statement

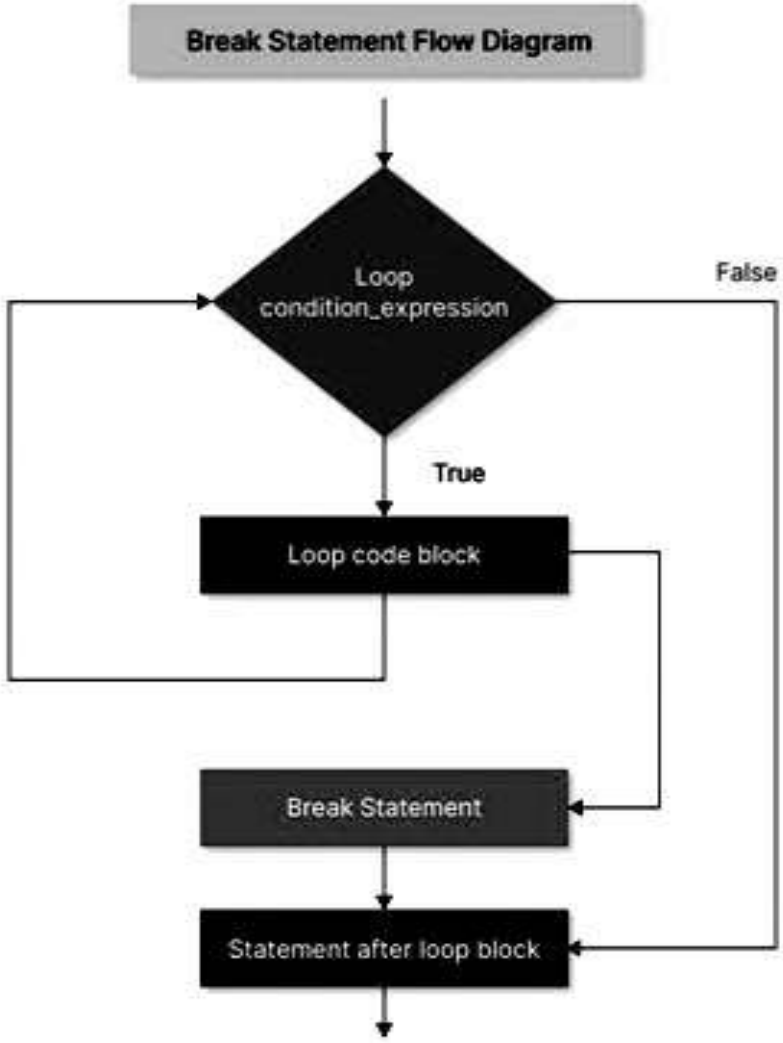


Fig. 10.7. Working of break Statement.

1. Loop condition is checked
2. Loop body starts executing
3. When break is encountered, the loop stops immediately
4. Control moves to the statement following the loop

Example of break in a Loop

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
}
```

Output

```
1
2
3
4
```

Here, the loop terminates when i becomes 5.

Use Cases of break

- Terminating search operations once a result is found
- Exiting a loop on error conditions
- Improving performance by avoiding unnecessary iterations

```
int[] nums = {2, 4, 6, 8, 10};
for (int n : nums) {
    if (n == 6) {
        break;
    }
    System.out.println(n);
}
```

Concept of continue Statement

The continue statement is used to skip the remaining statements of the current iteration and proceed directly to the next iteration of the loop. Unlike break, it does not terminate the loop entirely.

Syntax

```
continue;
```

Working of continue Statement

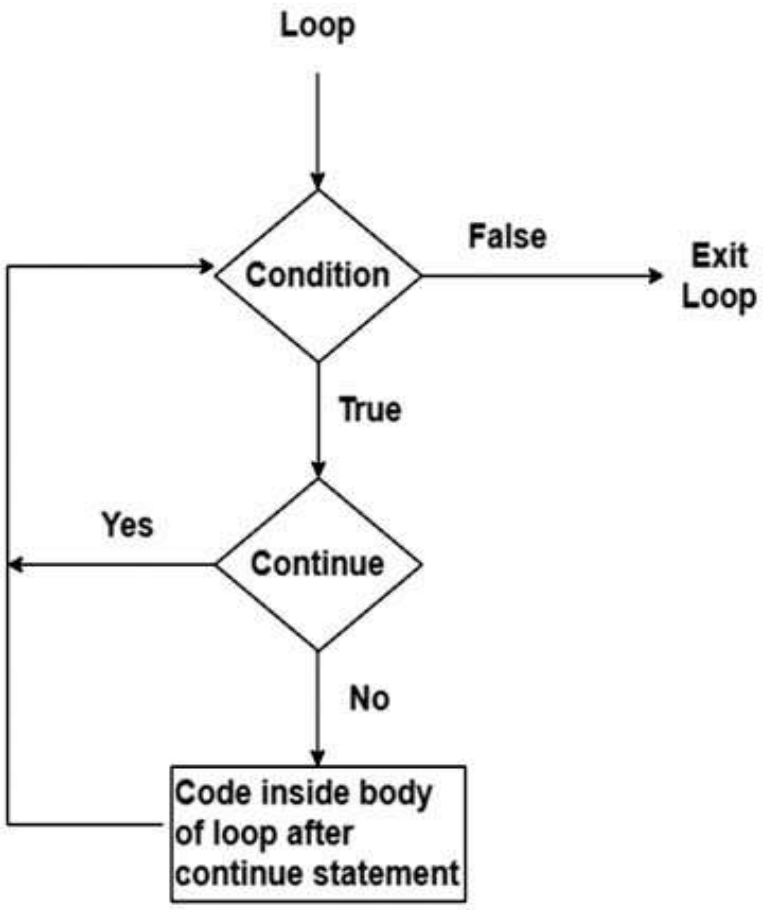


Fig. 10.8. Working of continue Statement.

NOTES

1. Loop condition is checked
2. Loop body executes
3. When continue is encountered, remaining statements are skipped
4. Control moves to the next iteration

Example of continue in a Loop

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;
    }
    System.out.println(i);
}
```

Output

1
2
4
5

The value 3 is skipped, but the loop continues.

Difference Between break and continue

Feature	break	continue
Effect	Terminates the loop	Skips current iteration
Control flow	Moves outside loop	Moves to next iteration
Use case	Exit loop early	Skip specific values

Common Mistakes

- Using break when continue is required
- Forgetting loop updates when using continue
- Overusing jump statements, reducing readability

Best Practices

- Use break for early termination only when necessary
- Use continue for clear filtering logic
- Avoid excessive use of jump statements
- Write clear comments when jump logic is complex

10.4.2. Labeled Loops

A labeled loop is a loop that is assigned an identifier followed by a colon (:). Labels are used in combination with break and continue to control outer loops from within nested loops. Labeled loops are

especially useful when dealing with nested loops, where a normal break or continue affects only the innermost loop.

Syntax

```
labelName:
for (...) {
// loop body
}
```

Labeled break Statement

A labeled break terminates the loop identified by the label.

Example

```
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (i == 2 && j == 2) {
            break outer;
        }
        System.out.println(i + " " + j);
    }
}
```

Explanation:
When the condition is met, execution jumps out of the **outer loop**, not just the inner one.

Labeled continue Statement

A labeled continue skips the current iteration of the labeled loop and continues with the next iteration of that loop.

Example

```
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            continue outer;
        }
        System.out.println(i + " " + j);
    }
}
```

Execution Flow of Labeled Loops

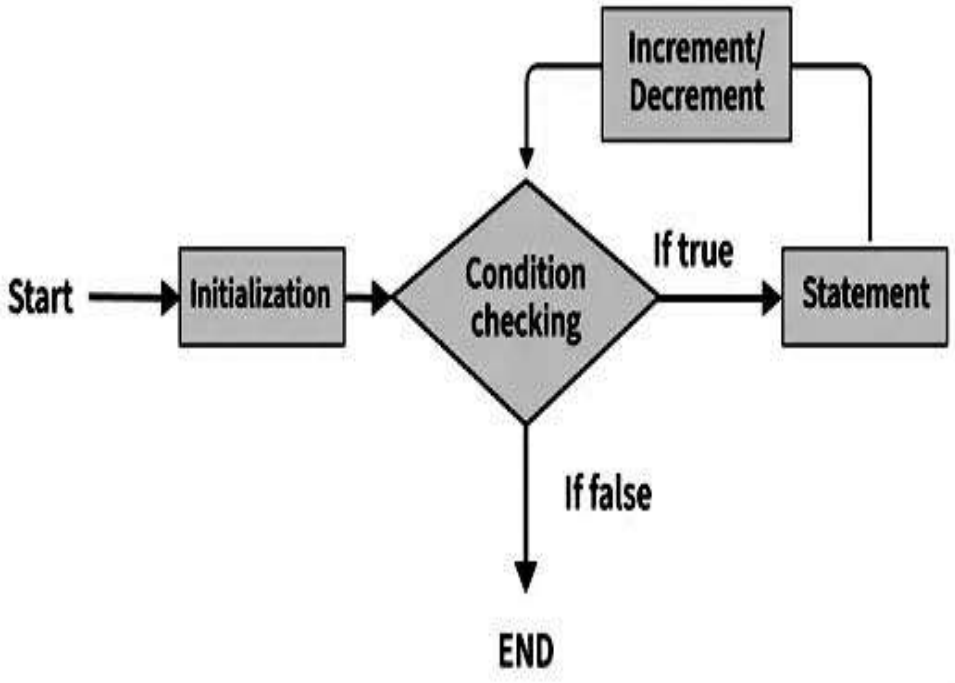


Fig. 10.9. Working of continue Statement.

Advantages of Labeled Loops

- Precise control over nested loops
- Eliminates need for additional flags
- Improves efficiency in complex looping logic

Disadvantages

- Reduces readability if overused
- Can confuse beginners
- Should be used sparingly

Best Practices for Labeled Loops

- Use meaningful label names
- Limit label usage to complex nested loops
- Add comments explaining label purpose
- Avoid deep nesting whenever possible

Activity 10.1 : Optimizing Processes with Loop Control

Objectives:
To evaluate how loop control statements improve efficiency.

Materials Needed:
Notebook and sample scenarios.

Activity Description:
Consider a customer database search system. Write a logical approach

where:

- The loop stops when a match is found (break)
- Certain records are skipped based on a condition (continue)

Explain how these controls improve performance.

Reflection/Outcome:
You will gain insight into optimizing algorithms and reducing unnecessary processing.

☑ Check Your Progress 10.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What are iteration statements in Java? Explain the for, while, and do–while loops.

   . . . . . .

   . . . . . .

   . . . . . .

2. What is the difference between break and continue statements?

   . . . . . .

   . . . . . .

   . . . . . .

3. What are labeled loops? How are they used in Java?

   . . . . . .

   . . . . . .

   . . . . . .

10.5. SUMMARY

This unit on **Control Statements – Looping** centers on developing structured, logical, and efficient programming practices in Java. It brings together key concepts that enhance the organization and clarity of code, enabling learners to manage program flow more effectively through disciplined use of loops. By engaging with these ideas, learners strengthen their analytical thinking and gain practical insight into handling repetitive tasks within programs. The unit emphasizes writing reliable and maintainable code, reducing logical and runtime errors while promoting better control over program execution. It also builds naturally on earlier programming concepts and lays a strong foundation for more advanced topics. The skills acquired here support modular and scalable program design, making them highly relevant in real-world Java applications. Overall, the unit contributes to both theoretical understanding and hands-on programming proficiency.

NOTES

10.6. Glossary

- **Program Structure** – The organized arrangement of statements in a program
- **Logic** – The reasoning that controls program execution
- **Execution Flow** – The order in which program statements are executed
- **Block** – A group of statements enclosed within braces
- **Condition** – An expression that evaluates to true or false
- **Iteration** – Repeated execution of a set of statements
- **Statement** – A single instruction in a program
- **Syntax** – Rules that define correct program structure
- **Modularity** – Dividing a program into smaller, manageable units
- **Maintainability** – Ease with which a program can be modified or updated.

10.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** Iteration statements are used to execute a block of code repeatedly.
   - The **for loop** is used when the number of iterations is known in advance.
   - The **while loop** executes if the condition is true and is used when the number of iterations is not fixed.
   - The **do–while loop** executes the block at least once before checking the condition, ensuring at least one execution.
2. **Answer :** The **break statement** is used to terminate a loop or switch statement immediately.

   The **continue statement** skips the current iteration and continues with the next iteration of the loop.
3. **Answer :** Labeled loops are loops that have a label assigned to them. They are used with break or continue to control outer loops in nested structures. This allows more precise control over loop execution, especially in nested loops.

10.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3751	

10.9. REFERENCES

1. Balagurusamy, E. *Programming with Java*. McGraw-Hill.
2. Schildt, H. *Java: The Complete Reference*. McGraw-Hill.
3. Deitel, P., & Deitel, H. *Java How to Program*. Pearson.
4. Oracle. *Java Documentation*.

10.10. EXERCISE

Part 1

A. Multiple Choice Questions

1. The main purpose of applying structured programming concepts in Java is to:

   (a) Increase program size

   (b) Reduce compilation time

   (c) Improve program clarity, logic, and maintainability

   (d) Avoid syntax rules

2. Proper use of programming constructs mainly helps in:

   (a) Increasing hardware performance

   (b) Reducing memory size

   (c) Managing program execution effectively

   (d) Avoiding variable declaration

3. A block of code in Java is defined using:

   (a) Parentheses ( )

   (b) Square brackets [ ]

   (c) Curly braces { }

   (d) Angle brackets < >

4. Program execution flow refers to:

   (a) Input and output methods

   (b) Memory allocation technique

   (c) Order in which program statements are executed

   (d) File handling mechanism

5. Logical correctness in a program ensures:

   (a) Faster compilation

   (b) More memory usage

   (c) Correct and expected program output

   (d) Larger program size

NOTES

Answers with Explanation :

1. Ans. (c)

Explanation : Structured programming helps organize code logically, making it easier to understand and maintain.

2. Ans. (c)

Explanation : Programming constructs control execution flow and decision making.

3. Ans. (c)

Explanation : Curly braces group multiple statements into a single executable block.

4. Ans. (c)

Explanation : Execution flow determines how instructions run step by step during program execution.

5. Ans. (c)

Explanation : Correct logic produces correct results and reliable program behavior.

B. Short Answer Questions (50–100 words each)

1. What is meant by structured programming in Java?
2. Explain the importance of execution flow in a program.
3. Define a block of code and state its significance.
4. Why is logical correctness essential in software development?
5. How does modular design improve the quality of Java programs?

Part 2

C. Long Answer Questions (200 words each)

1. Explain how structured program design improves readability and maintainability of Java programs.
2. Describe the role of logical organization in developing reliable and efficient software applications.

Part 3

D. Descriptive Answer Questions (300 words each)

1. Discuss the importance of logical thinking and disciplined program design in Java programming.
2. Explain how the concepts covered in this unit contribute to scalable, maintainable, and error-free software systems.

UNIT – 11
ARRAYS

UNIT STRUCTURE

11.1 Introduction
11.2 Objective
11.3 One-Dimensional Arrays
11.4 Multi-Dimensional Arrays
11.5 Summary
11.6 Glossary
11.7 Answer to check your progress
11.8 Suggested readings - OER
11.9 Reference
11.10 Exercise

11.1. INTRODUCTION

In earlier units, you studied variables, data types, and control statements, which allow programs to process data step by step. However, real-world programming problems rarely deal with a single value at a time. Instead, programs often need to handle collections of related data, such as marks of students, salaries of employees, temperatures of different cities, or monthly sales figures. Managing such data using individual variables becomes inefficient, error-prone, and difficult to maintain.

To solve this problem, programming languages like C and Java provide a powerful data structure known as an array. An array allows a program to store multiple values of the same data type under a single name and access them using an index. Among different types of arrays, the one-dimensional array is the simplest and most used form.

A one-dimensional array can be visualized as a linear list of elements stored in contiguous memory locations. Each element is uniquely identified by its index number, which makes accessing, updating, and processing data very efficient. Arrays are extensively used in tasks such as searching, sorting, statistical calculations, and data analysis. Understanding one-dimensional arrays is a foundation concept for advanced topics like two-dimensional arrays, strings, dynamic data structures, and collections. This unit introduces the concept of one-dimensional arrays, their declaration, initialization, memory representation, and common operations. By mastering arrays,

NOTES

students will be able to write compact, readable, and efficient programs suitable for academic and real-world applications.

11.2. OBJECTIVES

- After studying this unit, the learner will be able to:
- Define the concept of a one-dimensional array
- Identify the need for arrays in programming
- Declare and initialize one-dimensional arrays in programs
- Access and manipulate array elements using index values
- Analyze memory representation of arrays
- Develop simple programs using one-dimensional arrays

11.3. ONE-DIMENSIONAL ARRAYS

A one-dimensional array in Java is a structured data type that allows storage of multiple values of the same data type under a single variable name. Until now, programs written using primitive variables could store only one value per variable. However, real-world applications require handling large collections of related data, such as marks of students, temperatures of cities, or salaries of employees. Using individual variables for such data leads to complex and unreadable programs.

A one-dimensional array overcomes this limitation by organizing data in a linear sequence, where each element is stored in contiguous memory locations and accessed using an index. In Java, arrays are objects, which means they are created dynamically and managed by the Java Virtual Machine (JVM). This makes Java arrays safer and more flexible compared to arrays in low-level languages.

The index of a Java array always starts from 0, which means the first element is stored at index 0 and the last element is stored at length – 1. This indexing mechanism allows fast access to elements and efficient processing using loops. One-dimensional arrays are widely used in searching algorithms, sorting techniques, data analysis, and statistical computations. From a programming perspective, arrays significantly reduce code redundancy. For example, instead of writing multiple statements to process individual values, a loop can be used to process all array elements efficiently. Thus, understanding one-dimensional arrays is essential before moving to advanced concepts such as two-dimensional arrays, strings, collections, and data structures.

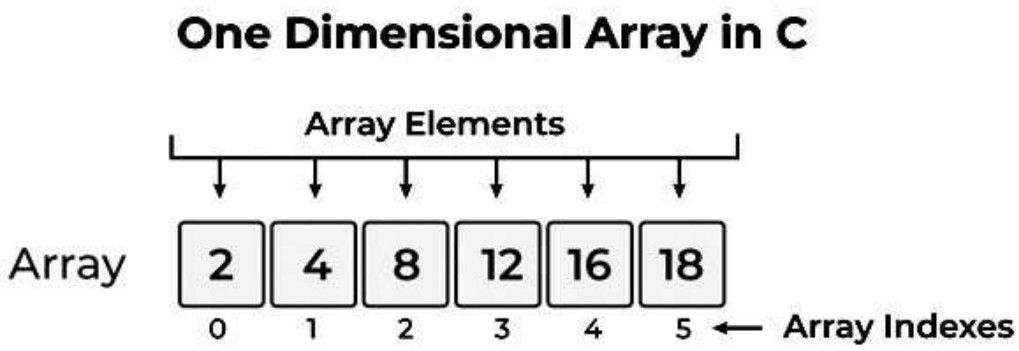


Fig. 11.1 : One Dimensional Array in C.

11.3.1. Array Declaration

Array declaration in Java specifies the type of data the array will store and the reference variable that will point to the array object. Unlike primitive variables, declaring an array does not immediately allocate memory for storing elements. It only creates a reference.

Syntax

```
dataType[] arrayName;
      or
dataType arrayName[];
```

Example

```
int[] marks;
float[] salary;
String[] names;
```

At this stage:

- Memory is **not allocated**
- The array reference is set to null
- The array cannot store values yet

To allocate memory, the new keyword must be used.

```
marks = new int[5];
```

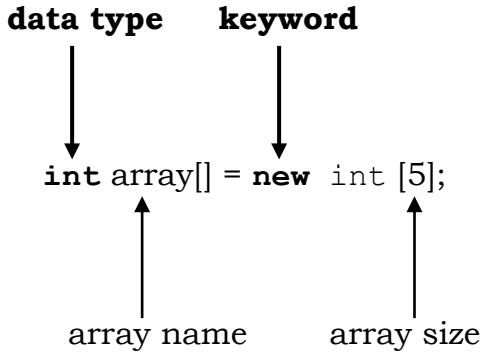
This statement:

- Creates an array of size 5
- Allocates contiguous memory
- Initializes elements with default values

Important Characteristics

- Array size is fixed once declared
- All elements must be of the same data type
- Java checks array index bounds at runtime, preventing illegal access

Array declaration is a critical step because it defines how much memory the program will reserve and how the data will be structured. Incorrect declaration can lead to runtime errors or inefficient memory usage.



11.3.2. Array Initialization

Array initialization refers to the process of assigning values to array elements. In Java, arrays can be initialized in multiple ways depending on program requirements.

1. Initialization at the Time of Declaration

```
int[] a = {10, 20, 30, 40, 50};
```

Here:

- Array size is automatically determined
- Values are stored sequentially
- Indexing starts from 0

2. Initialization Using new Keyword

```
int[] b = new int[5];
b[0] = 10;
b[1] = 20;
```

Unassigned elements get default values:

- int → 0
- float → 0.0
- boolean → false
- String → null

3. Initialization Using Loop

```
for (int i = 0; i < b.length; i++)
{
    b[i] = i * 2;
}
```

This method is useful when:

- Values are calculated dynamically
- Data is taken from user input
- Large datasets are involved

Array initialization ensures that data is ready for processing. Poor initialization may result in logical errors or incorrect output. Therefore, careful planning of array initialization is necessary in real-world Java applications.

11.4. MULTI-DIMENSIONAL ARRAYS

A multi-dimensional array is an array that contains other arrays as its elements. While one-dimensional arrays represent linear data, multi-dimensional arrays are used to represent tabular or matrix-like data. Examples include student result tables, seating arrangements, and mathematical matrices. In Java, multi-dimensional arrays are implemented as arrays of arrays, not as true matrices. This provides flexibility, as each row can have a different number of columns. Multi-dimensional arrays are essential in applications such as:

- Scientific computing
- Image processing
- Game development
- Data analysis and simulations

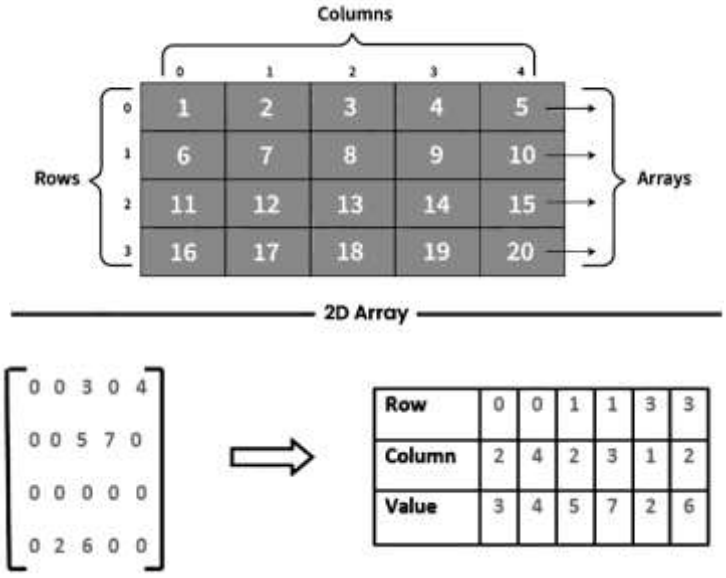


Fig. 11.2. Multi-Dimensional Arrays.

11.4.1. Two-Dimensional Arrays

A two-dimensional array is the most used multi-dimensional array. It represents data in rows and columns, like a table.

Syntax

```
dataType[][] arrayName = new dataType[rows][columns];
```

Example

```
int[][] marks = new int[3][4];
```

This creates:

- 3 rows
- 4 columns
- Total elements = 12

Accessing Elements

```
marks[0][0] = 85;
marks[1][2] = 90;
```

NOTES

Using Nested Loops

```
for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 4; j++) {
        System.out.print(marks[i][j] + " ");
    }
}
```

Two-dimensional arrays are widely used for:

- Marksheets
- Matrices
- Timetables
- Statistical tables

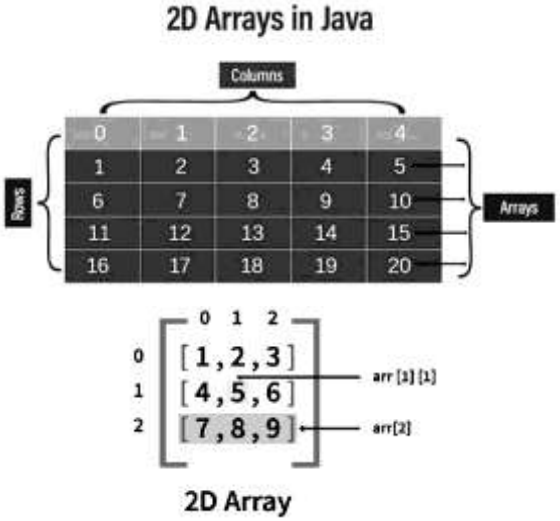


Fig. 11.3. Nested Loops.

```
1  public class NestedForLoopTemplate
2  {
3    public static void main(String args[])
4    {
5       int[] list = {3, 4, 11, 2, 7, 5};
6
7       for (int i = 0; i < list.length; i++){
8          for (int j = 0; j < list.length; j++){
9             System.out.print("possible pair(");
10            System.out.print(list[i]);
11            System.out.print(", ");
12            System.out.print(list[j]);
13            System.out.println(")");
14         }
15      }
16   }
17 }
```

11.4.2. MULTIDIMENSIONAL ARRAYS

A multidimensional array is an extension of the one-dimensional and two-dimensional array concepts, designed to store data that requires more than two levels of indexing. In Java, a multidimensional array is essentially an array of arrays, where each element of an array can itself be another array. This structure allows programmers to

represent complex real-world data in an organized and systematic manner.

In earlier sections, we studied one-dimensional arrays for linear data and two-dimensional arrays for tabular data such as matrices and tables. However, many practical applications involve data with three or more attributes. For example, consider storing examination marks of students for multiple subjects across different semesters. Such data cannot be efficiently represented using only one or two dimensions. Multidimensional arrays provide an effective solution for these scenarios.

In Java, multidimensional arrays are dynamic in nature, meaning memory allocation happens at runtime, and each dimension can be allocated independently. This makes Java arrays more flexible compared to traditional low-level languages. Understanding multidimensional arrays is important for applications in scientific computing, simulations, image processing, machine learning, and database-like data handling.

Concept of Multidimensional Arrays

A multidimensional array stores data in the form of layers. Each layer represents a dimension.

- A **one-dimensional** array represents a list
- A **two-dimensional** array represents a table
- A **three-dimensional** array represents a collection of tables
- Higher-dimensional arrays represent increasingly complex data structures

In Java, a multidimensional array does not need to be rectangular. This means different rows can have different column sizes, which is known as a jagged array.

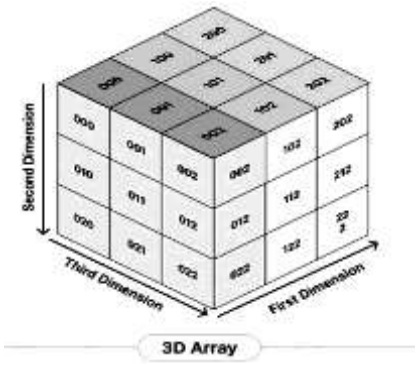


Fig. 11.4. Multidimensional Arrays.

Declaration of Multidimensional Arrays

The general syntax for declaring a multidimensional array in Java is:

```
dataType[][][] arrayName;
```

For example, declaring a three-dimensional integer array:

```
int[][][] marks;
```

At this stage:

NOTES

- Only the reference variable is created
- No memory is allocated
- The array cannot store values yet

Memory allocation is done using the new keyword:

```
marks = new int[2][3][4];
```

This creates:

- 2 blocks (e.g., semesters)
- Each block has 3 rows (e.g., students)
- Each row has 4 columns (e.g., subjects)

Thus, a total of 24 elements are created.

Initialization of Multidimensional Arrays

Multidimensional arrays can be initialized in several ways depending on program requirements.

1. Static Initialization

```
int[][][] data = {
{
{10, 20, 30},
{40, 50, 60}
},
{
{70, 80, 90},
{100, 110, 120}
}
};
```

In this method:

- Values are assigned at the time of declaration
- Size is determined automatically
- Data becomes immediately available for processing

This approach is suitable for small, fixed datasets.

2. Dynamic Initialization

```
int[][][] data = new int[2][2][3];
data[0][0][0] = 10;
data[1][1][2] = 50;
```

Advantages:

- Useful when data is generated at runtime
- Suitable for large datasets
- Supports user input and file-based input

All unassigned elements receive default values (0 for integers).

Accessing Elements in Multidimensional Arrays

Each element in a multidimensional array is accessed using multiple index values, one for each dimension.

```
data[block][row][column]
```

Example:

```
int value = data[1][0][2];
```

Here:

- First index selects the outer array
- Second index selects the inner array
- Third index selects the actual element

Incorrect indexing may result in ArrayIndexOutOfBoundsException, which Java checks at runtime, ensuring program safety.

Processing Multidimensional Arrays Using Loops

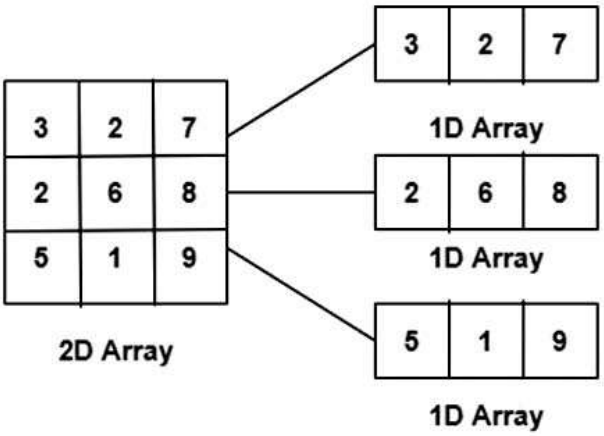
To process all elements of a multidimensional array, nested loops are required. The number of loops depends on the number of dimensions.

Example : Three-Dimensional Array Traversal

```
for(int i = 0; i < data.length; i++) {
    for(int j = 0; j < data[i].length; j++) {
        for(int k = 0; k < data[i][j].length; k++) {
            System.out.print(data[i][j][k] + " ");
        }
        System.out.println();
    }
}
```

This method ensures:

- Complete traversal of the array
- Efficient processing of large datasets
- Logical organization of data



Jagged Arrays (Irregular Multidimensional Arrays)

In Java, multidimensional arrays can be **jagged**, meaning each row can have a different number of columns.

Example

NOTES

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[4];
jagged[2] = new int[3];
```

Advantages of jagged arrays:

- Memory efficient
- Flexible data representation
- Suitable for uneven datasets

Jagged arrays are widely used in applications where data does not follow a strict tabular format.

Applications of Multidimensional Arrays

Multidimensional arrays are used in various domains:

- **Education Systems :** Marks of students across semesters and subjects
- **Scientific Computing :** Mathematical matrices and simulations
- **Image Processing :** Pixels represented as rows, columns, and color channels
- **Game Development :** Levels, maps, and object positions
- **Machine Learning :** Feature matrices and tensors

Their ability to model complex data structures makes them a crucial tool in advanced programming.

Advantages of Multidimensional Arrays

- Logical representation of complex data
- Efficient data organization
- Faster access using indexing
- Improved readability for structured data

Limitations of Multidimensional Arrays

- Complex syntax and indexing
- Increased memory usage
- Difficult debugging for beginners
- Reduced readability if dimensions are too many

Therefore, multidimensional arrays should be used only when the problem demands such structure.

Best Practices

- Use meaningful variable names
- Avoid unnecessary dimensions
- Validate array bounds carefully

- Prefer two-dimensional arrays where possible

Activity 11.1. Customer Feedback Rating Analysis

Objectives:
To develop analytical thinking by using arrays to evaluate customer satisfaction data.

Materials Needed:
Notebook/Spreadsheet tool

Activity Description:

1. Assume you collected feedback ratings (1–5 scale) from 10 customers. Store these ratings in a one-dimensional array.
2. Analyze the data by:
   - Counting how many ratings are above 3
   - Finding the frequency of each rating
   - Determining overall customer satisfaction level
3. Suggest one improvement strategy based on your findings.

Reflection/Outcome:
Learner gains insight into how structured data (arrays) can support business analysis and strategic planning. Reflect on how data-driven decisions improve service quality.

☑ Check Your Progress 11.1.

1. What is a one-dimensional array? Explain array declaration and initialization in Java.

   . . . . . .
   . . . . . .
   . . . . . .

2. What is a two-dimensional array? How is it different from a one-dimensional array?

   . . . . . .
   . . . . . .
   . . . . . .

3. What are multidimensional arrays? Explain their use in Java.

   . . . . . .
   . . . . . .
   . . . . . .

11.5. SUMMARY

Arrays are a fundamental concept in programming that allow multiple values to be stored under a single variable name, making data management more organized and efficient. A one-dimensional array uses a single index to access elements, and importantly, all elements

NOTES

within an array must belong to the same data type. Indexing typically begins from zero, which means the first element is accessed at position 0. Arrays are stored in contiguous memory locations, which enhances performance by enabling faster access and processing of data. Using loops with arrays simplifies repetitive tasks such as traversal, searching, and updating elements. Overall, arrays improve both program efficiency and readability by structuring data logically and reducing the need for multiple variables.

11.6. GLOSSARY

- **Array –** Collection of same-type elements
- **Index –** Position of an array element
- **Subscript –** Another term for index
- **Initialization –** Assigning values at declaration
- **Contiguous Memory –** Continuous memory allocation
- **Base Address –** Address of first array element
- **Loop –** Control structure for repetition
- **Out of Bounds –** Invalid index access
- **Element –** Individual data item in array
- **Traversal –** Accessing each element sequentially

11.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** A one-dimensional array is a collection of elements of the same data type stored in a linear form.
   - **Array declaration :** int arr[]; defines an array.
   - **Array initialization :** arr = new int[5]; or int arr[] = {1, 2, 3, 4, 5}; assigns values.
2. **Answer :** A two-dimensional array is an array of arrays, often represented in rows and columns like a table or matrix. For example, int arr[3][3];.

   It differs from a one-dimensional array in that it stores data in a tabular form rather than a single list.
3. **Answer :** Multidimensional arrays are arrays with more than two dimensions, such as 3D arrays. They are used to represent complex data structures like matrices, tables, or grids. They are declared as int arr[][][]; and help in organizing data in multiple levels.

11.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code

1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3752	

11.9. REFERENCES

- Balagurusamy, E. (2022). *Programming in ANSI C*. McGraw-Hill.
- Kanetkar, Y. (2021). *Let Us C*. BPB Publications.
- Schildt, H. (2020). *Java: The Complete Reference*. McGraw-Hill.
- NPTEL Online Courses.
- OpenStax Computer Science Resources.

11.10. EXERCISE

Part 1

A. Multiple Choice Questions

1. Which of the following is the correct syntax for declaring a one-dimensional array in C?

   (a) int arr;

   (b) int arr[10];

   (c) array int arr[10];

   (d) int[10] arr;

2. What is the default value of elements in an integer array if it is not initialized (in most cases)?

   (a) 1

   (b) Garbage value

   (c) 0

   (d) NULL

3. Which of the following represents correct initialization of an array?

   (a) int arr = {1,2,3};

   (b) int arr[3] = (1,2,3);

   (c) int arr[3] = {1,2,3};

   (d) array arr[3] = {1,2,3};

4. A two-dimensional array is also known as:

   (a) Linear array

   (b) Single array

   (c) Matrix

   (d) Pointer array

5. How is a two-dimensional array declared correctly?

   (a) int arr[3,4];

NOTES

(b) int arr(3)(4);

(c) int arr[3][4];

(d) array arr[3][4];

Answers with Explanations

1. Ans. (b)

Explanation : In C, a one-dimensional array is declared using the syntax data_type array_name[size]. Hence, int arr[10]; is correct.

2. (b) Garbage value

Explanation : If an array is declared locally and not initialized, its elements contain unpredictable (garbage) values.

3. Ans. (c)

Explanation : Arrays are initialized using curly braces {} with values separated by commas.

4. Ans. (c)

Explanation : A two-dimensional array represents data in rows and columns, similar to a matrix.

5. Ans. (c)

Explanation : A two-dimensional array is declared using two sets of square brackets, representing rows and columns respectively.

B. Short Answer Questions (50–100 words each)

1. Define one-dimensional array.
2. Write syntax for array declaration.
3. Explain array initialization.
4. What is contiguous memory?
5. Why are loops used with arrays?

Part 2

C. Long Answer Questions (200 words each)

1. Explain declaration, initialization, and access of one-dimensional arrays.
2. Describe memory representation of arrays with example.

Part 3

D. Descriptive Answer Questions (300 words each)

1. Discuss advantages and applications of one-dimensional arrays.
2. Explain array processing using loops with a complete program.

UNIT – 12
STRING HANDLING

UNIT STRUCTURE

12.1 Introduction

12.2 Objective

12.3 String Class

12.4 String Manipulation Methods

12.5 Summary

12.6 Glossary

12.7 Answer to check your progress

12.8 Suggested readings - OER

12.9 Reference

12.10 Exercise

12.1. INTRODUCTION

In the previous unit, students explored structured data handling techniques in Java, which strengthened their ability to organize, store, and process data efficiently. As learners progress further in Java programming, it becomes essential to consolidate foundational concepts while preparing for more advanced constructs. Unit 12 is designed to act as a transitional unit that reinforces disciplined programming practices and supports conceptual continuity within the course.

This unit focuses on aligning learner expectations with clearly defined academic goals and outcomes. A well-structured introduction helps students understand the relevance of the unit in the overall syllabus and how it connects with previously learned concepts. It also provides direction for upcoming topics, ensuring that learners approach the subject matter with clarity and confidence. The purpose of Unit 12 is not only to enhance technical knowledge but also to develop analytical thinking and structured problem-solving skills. Students are encouraged to reflect on prior learning, identify conceptual gaps, and actively engage with the learning process. Supporting components such as summaries, glossary, exercises, and suggested readings are included to reinforce understanding and promote independent learning.

NOTES

By the end of this unit, learners will be better prepared to comprehend advanced Java concepts, apply theoretical knowledge in practical scenarios, and approach programming problems methodically. Thus, Unit 12 serves as an important academic bridge that strengthens both conceptual understanding and application readiness.

12.2. OBJECTIVES

After completing this unit, the learner will be able to:

- **Recall** relevant Java programming concepts studied earlier
- **Identify** key terms and ideas related to the unit focus
- **Explain** the significance of structured learning components
- **Apply** learned concepts in basic programming situations
- **Analyze** program logic and outcomes related to the unit
- **Develop** systematic problem-solving skills using Java

12.3. STRING CLASS

In Java, strings are one of the most frequently used data types because almost every real-world application involves text processing. Java provides a built-in class called String, which is part of the java.lang package. Unlike primitive data types, a String is an object, and it represents a sequence of characters.

The String class provides a rich set of methods to create, manipulate, compare, and analyze strings. Java strings are designed with special features such as immutability, string pooling, and security, making them efficient and safe to use. Understanding the String class is essential for tasks such as user input handling, file processing, data validation, and application messaging.

Internally, a String object stores characters in a character array. However, Java hides these implementation details from the programmer, providing simple and powerful methods for string handling. Due to this object-oriented approach, strings in Java are more reliable and flexible than character arrays used in earlier programming languages.

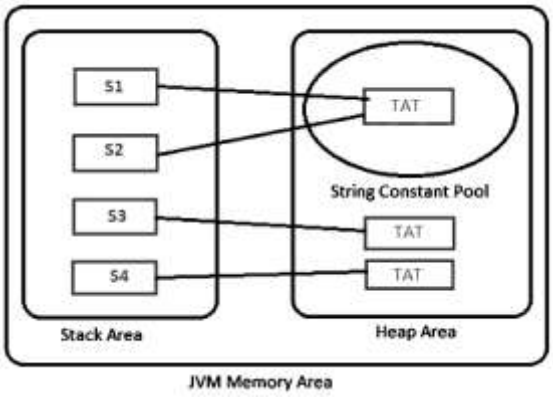


Fig. 12.1 (a) : string class.

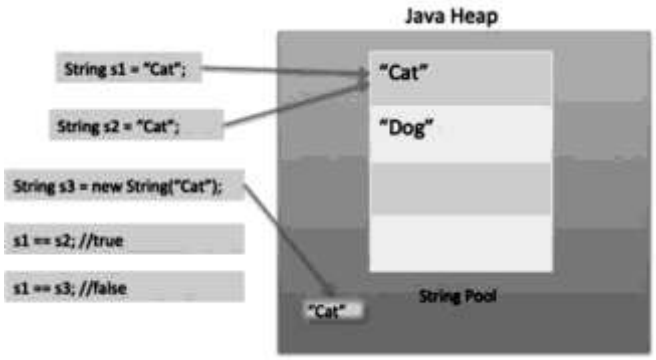


Fig. 12.1 (b) : string class.

12.3.1. String Creation

In Java, strings can be created in two main ways:

1. Using String Literal

String s1 = "Java";

- Stored in String Constant Pool
- Reuses existing objects
- Memory efficient

If another variable is assigned the same literal, both refer to the same object.

```
String s2 = "Java";
```

Here, s1 and s2 point to the same memory location.

2. Using new Keyword

```
String s3 = new String("Java");
```

- Creates a new object in heap memory
- Even if the same value exists, a new object is created

Comparison of String Creation Methods

Method	Memory Location	Object Creation
Literal	String Pool	Reused
new	Heap	Always new

String creation method affects memory usage, performance, and comparison behavior, making it an important design choice in Java applications.

12.3.2. String Concatenation

String concatenation is the process of combining two or more strings into a single string. Java supports multiple ways to concatenate strings.

1. Using + Operator

```
String s = "Hello" + " World";
```

- Simple and readable
- Automatically creates new String objects
- Not efficient in loops

NOTES

2. Using concat() Method

```
String s1 = "Java";
String s2 = s1.concat(" Programming");
```

- Returns a new string
- Original string remains unchanged

3. Using StringBuilder / StringBuffer

```
StringBuilder sb = new StringBuilder("Java");
sb.append(" Programming");
```

- Recommended for performance
- Reduces memory overhead
- Used in loops and large data processing

Performance Consideration

Using + repeatedly creates multiple objects, whereas StringBuilder modifies the same object, improving efficiency.

JavaExample.java

```
1  package com.beginnersbook;
2  public class JavaExample {
3      public static void main(String args[]) {
4          String mystring = ".com";
5          String mystr = "BeginnersBook".concat(mystring);
6          System.out.println(mystr);
7      }
8  }
```

Problems @ Javadoc Declaration Console Progress

```
<terminated> JavaExample [Java Application] /Library/Java/JavaVirtualMachines
BeginnersBook.com
```

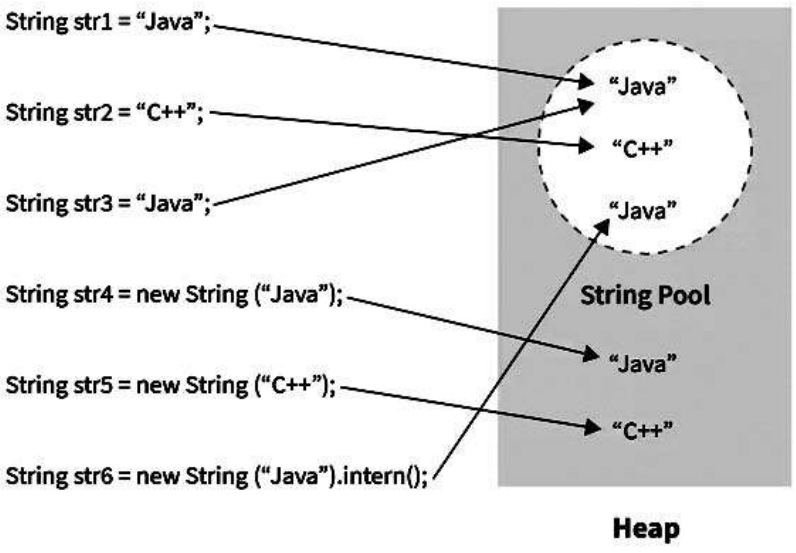


Fig. 12.2 : Performance Consideration.

12.3.3. String Immutability

One of the most important characteristics of Java strings is immutability. Once a String object is created, its value cannot be changed.

```
String s = "Java";
s = s.concat(" Language");
```

Here:

- A new string "Java Language" is created
- Original "Java" remains unchanged

Reasons for Immutability

1. **Security –** Prevents modification of sensitive data
2. **Thread Safety –** Safe for multi-threaded programs
3. **String Pool Optimization –** Enables reuse of objects
4. **Hashcode Caching –** Improves performance in collections

Mutable Alternatives

- StringBuffer – Thread-safe
- StringBuilder – Faster, not thread-safe

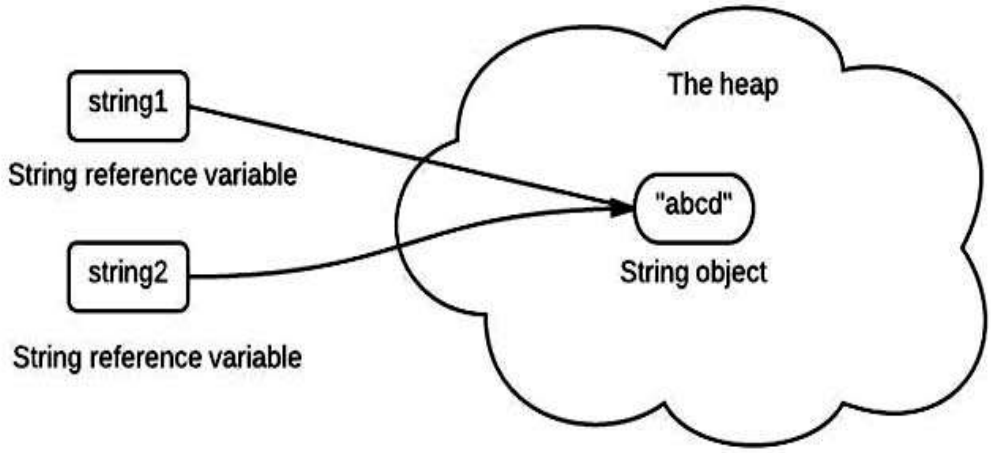


Fig. 12.3 : Mutable Alternatives

Summary of Differences

Feature	`String`	`StringBuffer`	`StringBuilder`
Mutability	Immutable	Mutable	Mutable
Thread-Safety	Thread-Safe	Thread-Safe (synchronized methods)	Not Thread-Safe (non-synchronized)
Performance	Lower performance in mutable operations due to object creation	Higher performance in mutable operations due to mutable nature	Higher performance in mutable operations due to non-synchronization
Memory Area	Heap Memory (String Pool)	Heap Memory (Internal mutable array)	Heap Memory (Internal mutable array)
Use Case	When string is constant or infrequent modifications	When string is modified in multi-threaded environment	When string is modified in single-threaded environment

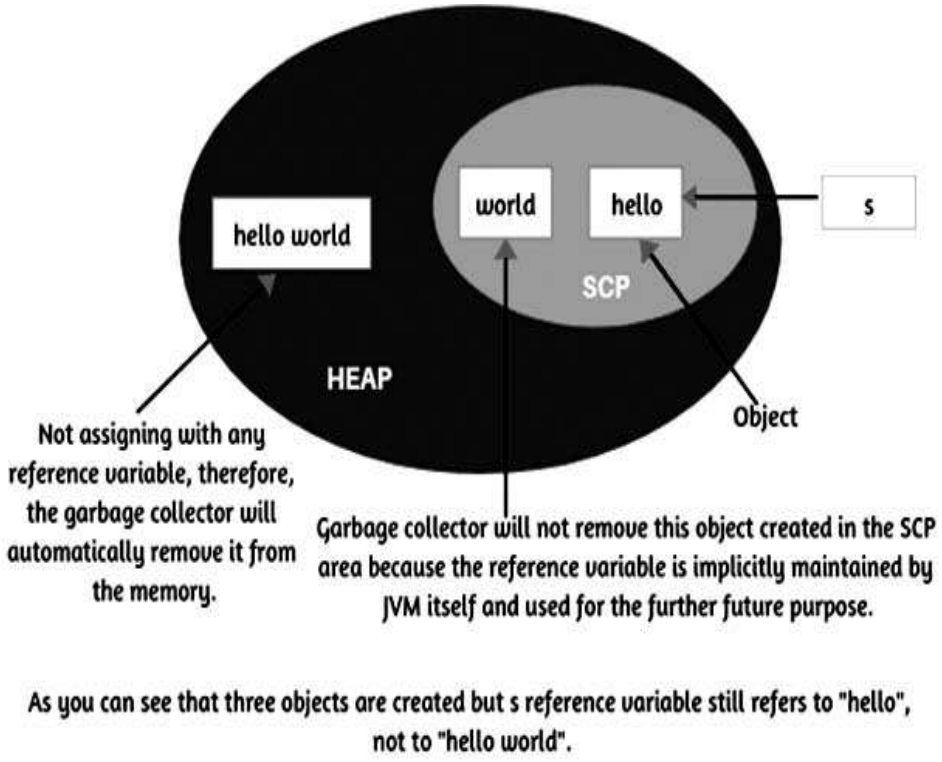


Fig. 12.4 : Allotting memory for storing object.

12.4. STRING MANIPULATION METHODS

Java provides many built-in methods in the String class to manipulate text efficiently. These methods help in comparison, searching, formatting, and conversion of string data.

String manipulation is essential in:

- User input validation
- Text processing
- File handling
- Web and database applications

12.4.1. String Comparison

String comparison determines whether two strings are equal or which one comes first lexicographically.

1. **equals() Method**

```
s1.equals(s2);
```

   - Compares content
   - Case-sensitive

2. **equalsIgnoreCase()**

```
s1.equalsIgnoreCase(s2);
```

   - Ignores case differences

3. **compareTo() Method**

```
s1.compareTo(s2);
```

- Returns 0 → equal
- Positive → s1 > s2
- Negative → s1 < s2

4. == Operator

- Compares reference, not content
- Not recommended for string comparison

Attribute	"=="	. equals ()
Type	Java binary Operator	Java Method of Object class
Compare logic	Compare variable and object references – memory location in Java heap	Compare the state or Content of objects – as per the logic in equals method
Override behaviour	Behaviour cannot be overridden	.equals() method can be overridden as it is java method. (String class cannot be extended as it is immutable class)
String Object Comparison	True, if objects are referring to same address	True- if objects hold the same String value.
Primitives	use "==" operator of equality check	Cannot use
Objects	True, if Objects pointed to same memory location.	Preferred to use equals() method . Ability to change the compare logic for equality
Objects created by new key word	Always return false though content are same.	True -if contents are same False if contents are not same

12.4.2. String Searching

Searching is used to locate characters or substrings inside a string.

Common Methods

```
indexOf()
lastIndexOf()
contains()
startsWith()
endsWith()
```

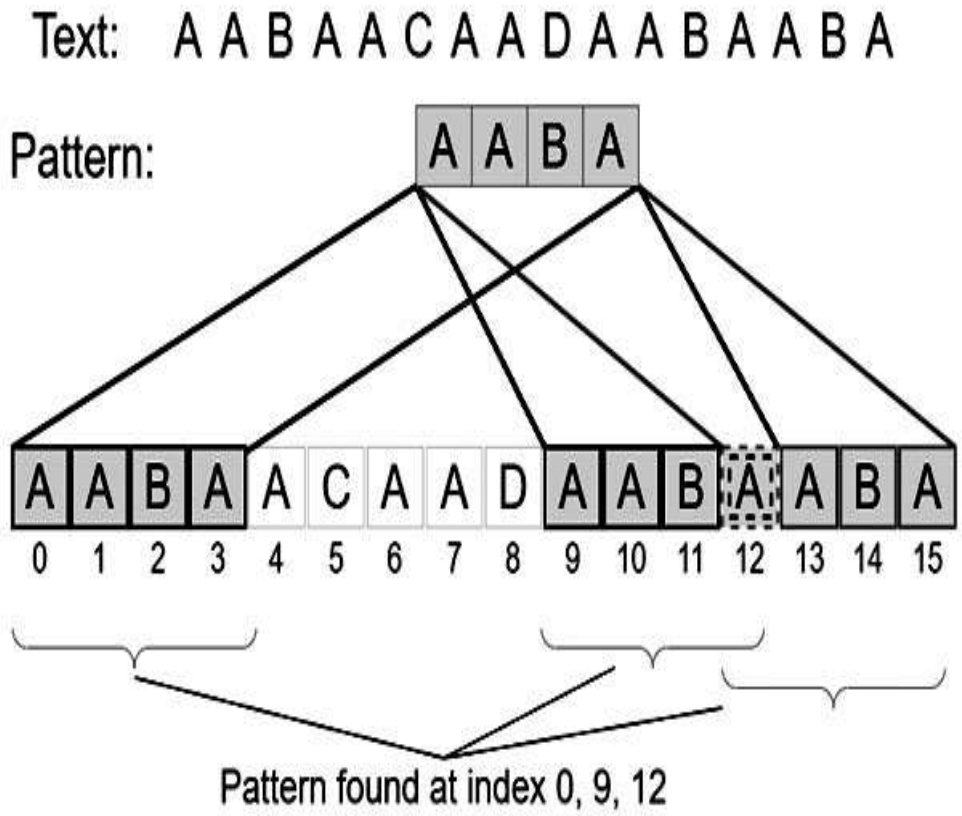
Example

```
String s = "Java Programming";
s.indexOf("Pro");   // returns 5
```

Applications:

- Keyword detection
- Input validation
- Text analysis

NOTES



12.4.3. Case Conversion

Java allows conversion of string case to support formatting and normalization.

Methods

```
toUpperCase()
toLowerCase()
```

Example

```
String s = "Java";
System.out.println(s.toUpperCase());
```

Case conversion is widely used in:

- Data validation
- Search operations
- User interface formatting

Activity 12.1.

Write a program to perform string operations like length, substring, and comparison. Observe how strings are manipulated.

NOTES

☑ **Check Your Progress 12.1.**

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What is a String in Java? Explain string creation and concatenation.

. . . . . .

. . . . . .

. . . . . .

2. What is string immutability? Why is it important in Java?

. . . . . .

. . . . . .

. . . . . .

3. Explain different string manipulation methods such as comparison, searching, and case conversion.

. . . . . .

. . . . . .

. . . . . .

12.5. SUMMARY

- This unit reinforces continuity between previously learned concepts and upcoming Java topics
- Clear objectives guide learners toward measurable learning outcomes
- Emphasis is placed on structured thinking and disciplined coding practices
- The unit encourages reflection on prior knowledge to improve conceptual clarity
- Supporting sections such as glossary and exercises enhance understanding
- Self-assessment through questions helps learners evaluate their progress
- Suggested readings promote independent and extended learning
- Overall, the unit strengthens the academic foundation required for advanced Java programming

12.6. GLOSSARY

- **Objective –** A clearly defined learning outcome of a unit
- **Conceptual Clarity –** Clear understanding of programming ideas
- **Syntax –** Rules that define the structure of a program

- **Logic –** Step-by-step reasoning used to solve problems
- **Compilation –** Conversion of source code into executable form
- **Execution –** Running a program to produce output
- **Error –** A mistake that causes incorrect program behavior
- **Debugging –** Process of finding and fixing errors
- **Algorithm –** A finite set of steps to solve a problem
- **Program Flow –** Order in which instructions are executed

12.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** A String in Java is a sequence of characters used to represent text.
   - **String creation :** Strings can be created using string literals (String s = "Hello";) or using the new keyword (String s = new String("Hello");).
   - **String concatenation :** It is the process of joining two or more strings using the + operator or the concat() method.
2. **Answer :** String immutability means that once a String object is created, its value cannot be changed. Any modification results in the creation of a new object. This improves security, memory efficiency, and performance (due to string pooling).
3. **Answer :** String manipulation methods include:
   - **String comparison :** Methods like equals() and compareTo() are used to compare strings.
   - **String searching :** Methods like indexOf() and contains() are used to find characters or substrings.
   - **Case conversion :** Methods like toUpperCase() and toLowerCase() are used to change the case of characters in a string.

12.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3753	

12.9. REFERENCES (APA 7TH EDITION)

- Balagurusamy, E. (2022). *Programming with Java*. McGraw-Hill Education.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Oracle. (2023). *Java Documentation*.
- NPTEL. (2022). *Java Programming Courses*.

12.10. EXERCISE

Part 1

A. Multiple Choice Questions

1. In Java, a String is:
   (a) A primitive data type
   (b) A sequence of characters
   (c) A numeric value
   (d) A loop structure
2. Which of the following is a correct way to create a String?
   (a) String s = Hello;
   (b) String s = "Hello";
   (c) string s = "Hello";
   (d) s = new String Hello;
3. String concatenation is performed using:
   (a) - operator
   (b) * operator
   (c) + operator
   (d) / operator
4. String immutability means:
   (a) Strings can be modified anytime
   (b) Strings cannot be changed after creation
   (c) Strings are deleted automatically
   (d) Strings are always numeric
5. Which method is used for string comparison?

(a) equals()

(b) add()

(c) insert()

(d) append()

Answers with Explanations

1. Ans. (b)

String Concept : A String in Java is a sequence of characters used to represent text.

2. Ans. (b)

String Creation : Strings are created using double quotes, such as String s = "Hello";

3. Ans. (c)

String Concatenation : The + operator is used to join two or more strings into a single string.

4. Ans. (b)

String Immutability: Strings cannot be modified after creation; any change creates a new object.

5. Ans. (a)

String Comparison : The equals() method is used to compare the content of two strings.

B. Short Answer Questions (50–100 words)

1. Explain the importance of unit objectives.
2. Describe the role of a glossary in SLM.
3. Why are summaries essential in academic units?
4. What is the purpose of suggested readings?
5. Define debugging.

Part 2

C. Long Answer Questions (200 words)

1. Discuss how structured objectives improve student learning.
2. Explain the importance of exercises in self-learning materials.

Part 3

D. Long Answer Questions (300 words)

1. Explain how SLM structure supports independent learning in Java programming.
2. Discuss the role of summaries, glossary, and exercises in reinforcing concepts.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | DCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC | A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

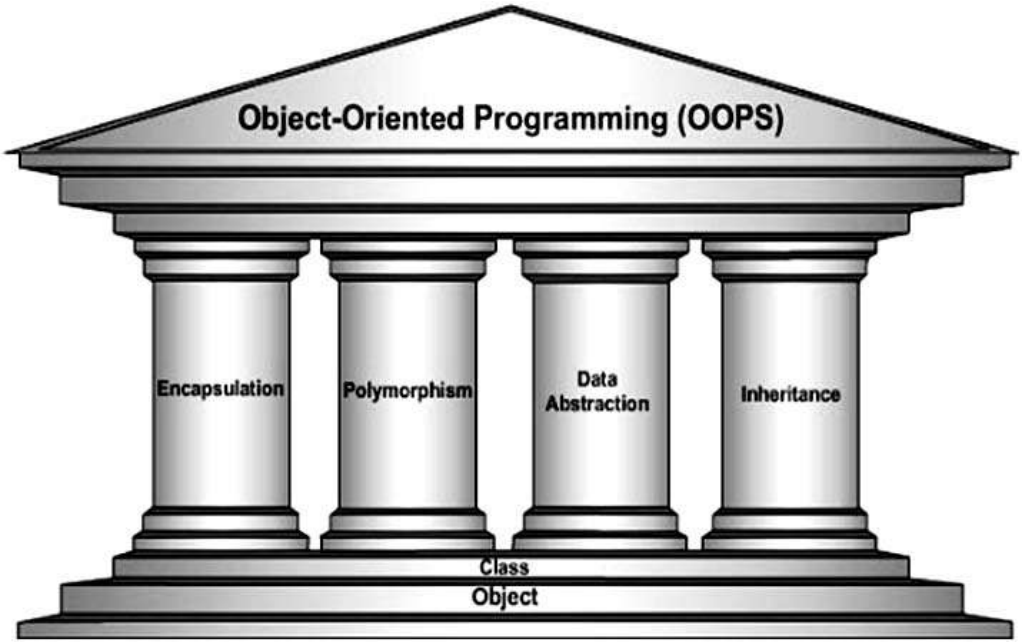
_____

_____

_____

***

Block V	OBJECT-ORIENTED BUILDING BLOCKS	179 - 222
Unit 13	StringBuffer and Wrapper Classes	181
Unit 14	Classes and Objects	193
Unit 15	Methods and Constructors	209



Object-oriented programming is built on fundamental building blocks that help in designing structured and reusable software. Classes and objects form the core of this approach, where classes act as blueprints and objects represent real-world entities. Methods define the behavior of objects, while constructors are used to initialize them. In addition, special classes like StringBuffer support dynamic string manipulation, and wrapper classes enable primitive data types to be treated as objects. Together, these concepts promote modularity, flexibility, and efficient problem-solving, making object-oriented programming a powerful paradigm for developing robust and scalable applications.

UNIT – 13
STRINGBUFFER AND WRAPPER CLASSES

UNIT STRUCTURE

13.1 Introduction

13.2 Objective

13.3 StringBuffer Class

13.4 Wrapper Classes

13.5 Summary

13.6 Glossary

13.7 Answer to check your progress

13.8 Suggested readings - OER

13.9 Reference

13.10 Exercise

13.1. INTRODUCTION

In the previous unit, learners were introduced to string handling concepts in Java, which play a crucial role in text processing and real-world application development. As students advance further in Java programming, it becomes important to consolidate earlier learning while gradually preparing for more complex programming constructs. Unit 13 is designed to provide this continuity by strengthening conceptual understanding and encouraging structured thinking.

This unit emphasizes the importance of clarity in learning objectives, systematic revision of concepts, and the ability to apply knowledge in a logical manner. Programming is not only about writing correct syntax but also about understanding program behavior, flow of execution, and efficient problem-solving strategies. Therefore, this unit supports learners in developing analytical skills alongside technical competence.

The structure of Unit 13 follows a self-learning approach where students are guided through clearly defined objectives, summarized learning outcomes, and supportive learning tools such as glossary, exercises, and suggested readings. These components help learners assess their understanding, revise key ideas, and explore additional resources for deeper learning.

NOTES

By engaging with this unit, students are expected to gain confidence in applying Java programming concepts, improve their ability to analyze program logic, and prepare themselves for more advanced topics in subsequent units. Thus, Unit 13 serves as an essential academic step that strengthens both theoretical understanding and practical readiness in Java programming.

13.2. OBJECTIVES

After completing this unit, the learner will be able to:

- **Recall** relevant Java programming concepts studied earlier
- **Identify** key ideas and terms associated with this unit
- **Explain** the importance of structured learning components in programming
- **Apply** learned concepts to simple Java programming scenarios
- **Analyze** program flow and logic related to the unit focus
- **Develop** disciplined problem-solving skills using Java

13.3. STRINGBUFFER CLASS

In Java programming, strings are widely used for handling textual data. In the previous unit, students studied the Stringclass and learned that strings in Java are immutable, meaning their values cannot be changed once created. While immutability provides advantages such as security and thread safety, it can also lead to performance issues when frequent modifications to strings are required. To overcome this limitation, Java provides the StringBuffer class.

The StringBuffer class is a part of the java.lang package and is used to create mutable strings. A mutable string means that the content of the object can be changed without creating a new object. This feature makes StringBuffer particularly useful in situations where strings need to be modified repeatedly, such as in loops, file processing, or dynamic text generation. Unlike the String class, which creates a new object whenever a modification occurs, StringBuffer modifies the existing object in memory. This reduces memory overhead and improves execution efficiency. Another important feature of StringBuffer is that it is thread-safe, meaning it is synchronized and safe to use in multi-threaded environments.

Internally, StringBuffer maintains a dynamically resizable character array. When the capacity of this array is exceeded, it automatically increases its size, allowing more characters to be added. This dynamic behavior makes StringBuffer suitable for handling large and changing strings.

Thus, the StringBuffer class plays a crucial role in Java applications that demand high performance, frequent string modifications, and thread safety.

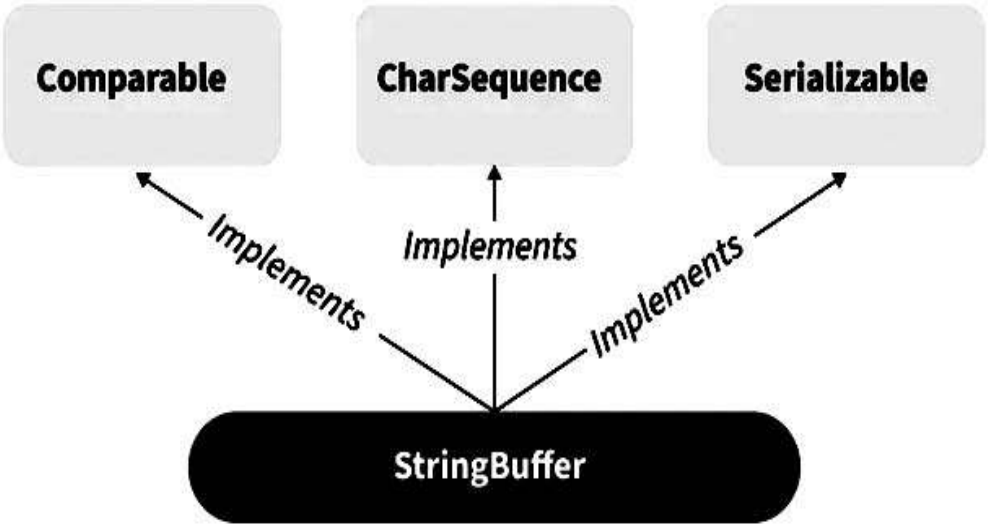


Fig. 13.1 : StringBuffer.

13.3.1. StringBuffer Methods

The StringBuffer class provides a wide range of methods that allow efficient manipulation of string data. These methods operate directly on the same object, making them faster compared to String methods.

append() Method

The append() method adds data to the end of the existing string buffer.

```
StringBuffer sb = new StringBuffer("Java");
sb.append(" Programming");
```

Output : Java Programming

This method can append different data types such as integers, floats, characters, and strings.

insert() Method

The insert() method inserts data at a specified position.

```
sb.insert(4, " Language");
```

```
delete() and deleteCharAt() Methods
```

These methods remove characters from the string buffer.

```
sb.delete(0, 4);
sb.deleteCharAt(2);
```

reverse() Method

The reverse() method reverses the sequence of characters.

```
sb.reverse();
```

capacity() and length() Methods

- length() returns the number of characters
- capacity() returns the total allocated space

These methods help in memory management and optimization.

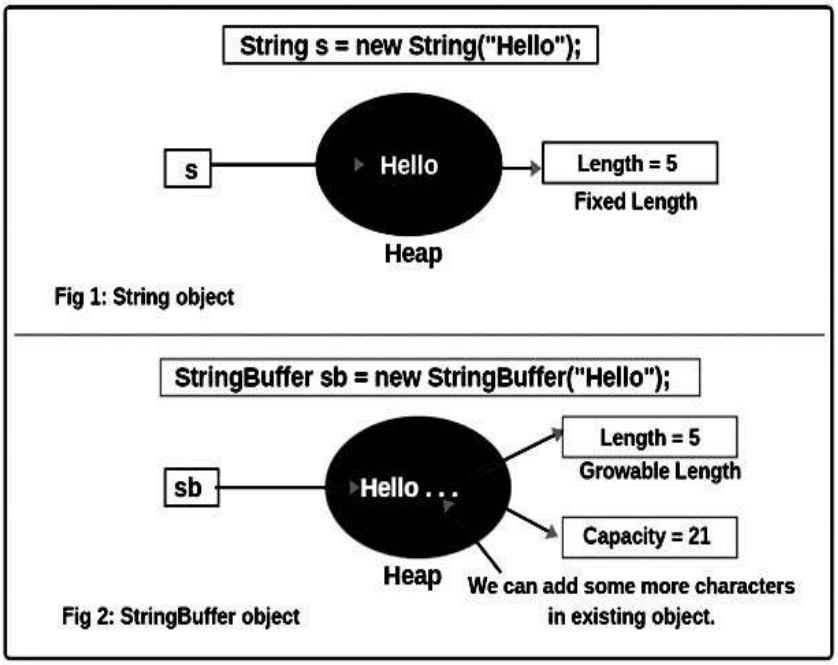


Fig. 13.2. SringBuffer Object

13.3.2. Performance Advantages

The main advantage of StringBuffer over String lies in its **performance efficiency** during frequent string modifications.

Why StringBuffer is Faster

- No creation of new objects on modification
- Reduced memory consumption
- Efficient handling in loops
- Synchronized for thread safety

Comparison with String

Feature	String	StringBuffer
Mutability	Immutable	Mutable
Thread Safety	Yes	Yes
Performance	Slower	Faster
Memory Usage	High	Optimized

StringBuffer is widely used in:

- Multi-threaded applications
- File and data processing
- Logging systems

However, in single-threaded applications where thread safety is not required, StringBuilder is often preferred due to better performance.

13.4. WRAPPER CLASSES

In Java, primitive data types such as int, float, char, and Boolean are not objects. However, Java is an object-oriented language, and many features such as collections, generics, and frameworks require objects instead of primitives. To bridge this gap, Java provides Wrapper Classes.

A wrapper class is a class that encapsulates a primitive data type into an object. All wrapper classes are present in the java.lang package. Wrapper classes allow primitive values to be treated as objects, enabling them to be used in collections like ArrayList, HashMap, and other object-oriented contexts. Java also supports autoboxing (automatic conversion of primitive to object) and unboxing (conversion of object to primitive), which simplifies coding and improves readability.

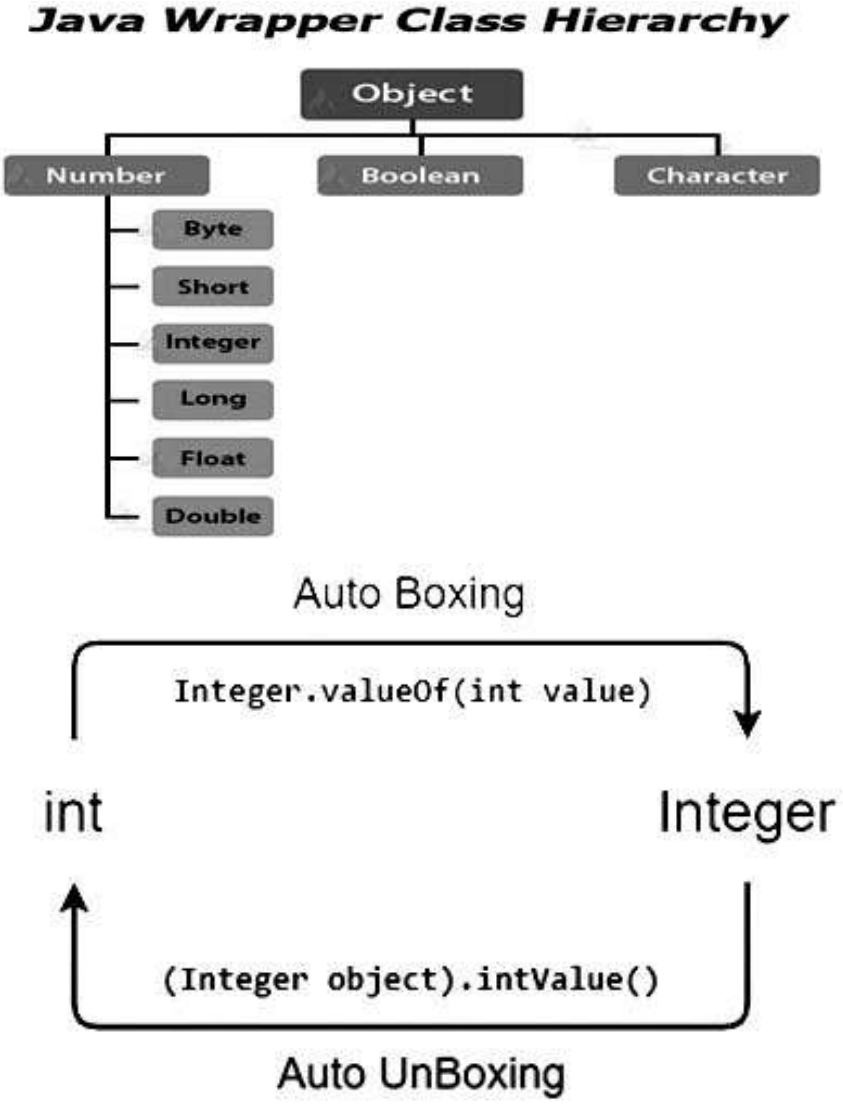


Fig. 13.4 : Wrapper Classes.



Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

13.4.1. Numeric Wrapper Classes

Numeric wrapper classes are used to wrap numeric primitive data types.

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Example :

```
Integer x = 10;    // Autoboxing
int y = x;         // Unboxing
```

Common Methods

- `parseInt()`
- `valueOf()`
- `compareTo()`

These methods help in data conversion, comparison, and validation.

Numeric wrapper classes are widely used in:

- Collections
- Database operations
- Input validation

13.4.2. Character Wrapper Class

The Character class wraps the primitive char data type. It provides several utility methods for character manipulation and analysis.

Example :

```
Character ch = 'A';
```

Common Methods

- `isLetter()`
- `isDigit()`
- `isUpperCase()`
- `toLowerCase()`

These methods are useful in:

- `Input validation`
- `Text parsing`
- `Lexical analysis`

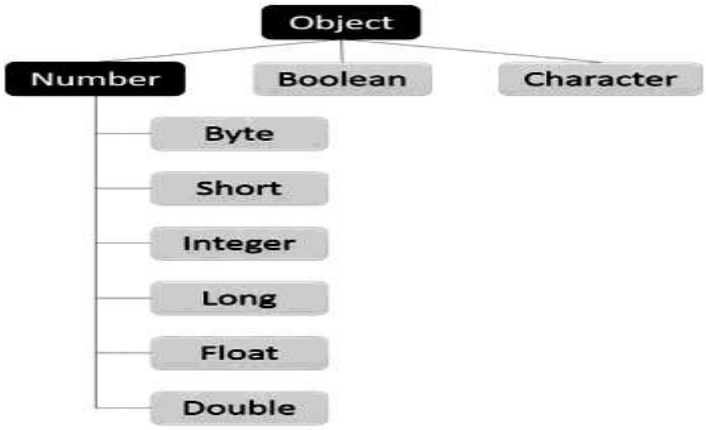


Fig. 13.6 : Character Wrapper Class.

13.4.3. Boolean Wrapper Class

In Java, the primitive data type Boolean is used to represent logical values, which can be either true or false. Logical values play a crucial role in programming because they control decision-making, conditional execution, and looping structures. However, primitive data types like Boolean are not objects and therefore cannot be directly used in object-oriented features such as collections, generics, or frameworks. To overcome this limitation, Java provides the Boolean wrapper class.

The Boolean class is a part of the java. lang package and acts as a wrapper for the primitive Boolean data type. It allows Boolean values to be treated as objects, enabling their use in object-based programming environments. The Boolean class encapsulates a single Boolean value and provides several useful methods for conversion, comparison, and logical evaluation.

One of the major advantages of the Boolean wrapper class is its support for autoboxing and unboxing. Autoboxing automatically converts a primitive Boolean into a Boolean object, while unboxing converts a Boolean object back into a primitive Boolean. This feature simplifies code and improves readability by reducing the need for explicit conversions.

```
Boolean b1 = true;   // Autoboxing
boolean b2 = b1;     // Unboxing
```

NOTES

In real-world Java applications, Boolean objects are commonly used in configuration settings, feature flags, decision variables, and conditional parameters, especially when values need to be stored in collections or passed as objects between methods.

Creation of Boolean Objects

Boolean objects can be created in two main ways:

1. Using Boolean Constructor

```
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean("false");
```

In this approach:

- A new Boolean object is created
- The constructor accepts either a Boolean value or a string
- If the string is not equal to "true" (ignoring case), the value is set to false

2. Using valueOf() Method (Recommended)

```
Boolean b3 = Boolean.valueOf(true);
Boolean b4 = Boolean.valueOf("TRUE");
```

The valueOf() method is preferred because it may reuse existing objects, improving performance and memory efficiency.

Common Methods of Boolean Class

The Boolean wrapper class provides several useful methods:

booleanValue()

Returns the primitive Boolean value stored in the Boolean object.

```
Boolean b = Boolean.TRUE;
boolean result = b.booleanValue();
parseBoolean()
```

Converts a string into a primitive boolean.

```
boolean flag = Boolean.parseBoolean("true");
```

This method returns true only if the string is "true" (case-insensitive); otherwise, it returns false.

toString()

Returns the string representation of the Boolean value.

```
String s = b.toString();
```

equals()

Compares two Boolean objects for logical equality.

```
b1.equals(b2);
```

Boolean Constants

The Boolean class provides two predefined constants:

```
Boolean.TRUE
Boolean.FALSE
```

These constants improve code readability and reduce the need to create new objects.

Applications of Boolean Wrapper Class

The Boolean wrapper class is widely used in:

- **Collections** such as ArrayList<Boolean> and HashMap<String, Boolean>
- **Configuration flags** in applications
- **Conditional logic** passed as method arguments
- **Frameworks and APIs** that require objects instead of primitives
- **Database interaction**, where values are retrieved as objects

Advantages of Boolean Wrapper Class

- Enables use of Boolean values in object-oriented contexts
- Supports autoboxing and unboxing
- Provides utility methods for conversion and comparison
- Improves code clarity in complex applications

Limitations

- Slightly more memory usage compared to primitive Boolean
- Object creation overhead if misused
- Should not replace primitive Boolean in simple conditions

Activity 13.1.

Compare String and StringBuffer through a small program. Use wrapper classes to convert data types.

☑ Check Your Progress 13.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What is the StringBuffer class and why is it used?

. . . . . .

. . . . . .

. . . . . .

2. Explain the performance advantages of StringBuffer over String.

. . . . . .

. . . . . .

. . . . . .

3. What are wrapper classes? Give examples of numeric, character, and boolean wrapper classes.

. . . . . .

. . . . . .

. . . . . .

13.5. SUMMARY

This unit on **StringBuffer and Wrapper** Classes serves as a thoughtful bridge between foundational Java concepts and more advanced programming topics. It is carefully structured with clearly defined objectives that guide learners toward meaningful outcomes, while maintaining a strong focus on logical thinking and well-organized program design. By encouraging learners to revisit and reflect on previously learned concepts, the unit deepens conceptual clarity and strengthens understanding. The inclusion of tools such as glossaries, exercises, and self-assessment questions supports effective revision and allows learners to monitor their own progress. Additionally, suggested readings extend learning beyond the core material, fostering independence and intellectual curiosity. Overall, this unit plays a crucial role in building a solid academic and practical foundation, preparing learners to confidently engage with more complex Java programming concepts ahead.

13.6. GLOSSARY

- **Concept Reinforcement –** Strengthening previously learned ideas
- **Objective –** A measurable learning outcome of a unit
- **Syntax –** Rules governing the structure of Java programs
- **Logic –** Step-by-step reasoning used to solve problems
- **Execution Flow –** Order in which program statements run
- **Compilation –** Process of converting source code into bytecode
- **Runtime –** Phase during which a program is executed
- **Error –** A mistake that causes incorrect program behavior
- **Debugging –** Identifying and correcting errors in programs
- **Self-Learning Material (SLM) –** Structured content designed for independent study

13.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** The StringBuffer class in Java is used to create mutable (modifiable) string objects. Unlike the String class, its content can be changed without creating a new object. It is mainly used when frequent modifications are required.
2. **Answer :** StringBuffer provides better performance than String when **multiple** modifications are needed. This is because it does not create new objects for every change, thereby saving memory and improving efficiency. It is also thread-safe, making it suitable for multi-threaded environments.
3. **Answer :** Wrapper classes are used to convert primitive data types into objects. Examples include:
   - Numeric wrapper classes: Integer, Double, Float
   - Character wrapper class: Character

- Boolean wrapper class: Boolean

These classes allow primitives to be used in collections and support object-oriented features.

13.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3754	

13.9. REFERENCES

- Balagurusamy, E. (2022). *Programming with Java*. McGraw-Hill Education.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Oracle. (2023). *Java Documentation*.
- NPTEL. (2022). *Java Programming Courses*.

13.10. EXERCISE

Part 1

A. Multiple Choice Questions

1. StringBuffer class in Java is:
   (a) Immutable (b) Mutable
   (c) Static (d) Final
2. Which method is used to add content to a StringBuffer?
   (a) add() (b) append()
   (c) concat() (d) insertData()
3. The main advantage of StringBuffer over String is:
   (a) It is immutable
   (b) It improves performance during frequent modifications
   (c) It reduces program size
   (d) It avoids object creation
4. Wrapper classes are used to:
   (a) Convert objects into primitives
   (b) Convert primitives into objects
   (c) Store arrays
   (d) Execute loops
5. Which of the following is a Boolean wrapper class?
   (a) Integer
   (b) Character

NOTES

(c) Boolean

(d) Double

Answers with Explanations

1. Ans. (b)

StringBuffer Nature : StringBuffer is mutable, meaning its content can be modified without creating a new object.

2. Ans. (b)

StringBuffer Method : The append() method is used to add data to an existing StringBuffer object.

3. Ans. (b)

Performance Advantage : StringBuffer improves performance when frequent modifications are required because it avoids creating multiple objects.

4. Ans. (b)

Wrapper Classes Purpose : Wrapper classes convert primitive data types into objects, enabling their use in collections and object-oriented features.

5. Ans. (c)

Boolean Wrapper Class : The Boolean class is used to represent true/false values as objects.

B. Short Answer Questions (200 words)

1. Explain the importance of objectives in an SLM unit.
2. Describe the role of a glossary in learning Java.
3. Why are summaries important in academic units?
4. What is the purpose of suggested readings?
5. Define debugging.

Part 2

C. Long Answer Questions (300 words)

1. Discuss how structured objectives support effective learning in Java.
2. Explain the role of exercises in self-learning materials.

Part 3

D. Long Answer Questions (300 words)

1. Explain how SLM structure helps students become independent learners.
2. Discuss the importance of summaries, glossary, and exercises in reinforcing programming concepts.

UNIT – 14
CLASSES AND OBJECTS

UNIT STRUCTURE

14.1 Introduction
14.2 Objective
14.3 Defining Classes and Objects
14.4 Accessing Class Members
14.5 Summary
14.6 Glossary
14.7 Answer to check your progress
14.8 Suggested readings - OER
14.9 Reference
14.10 Exercise

14.1. INTRODUCTION

In the earlier units, learners have gradually built a strong foundation in Java programming by studying arrays, strings, string manipulation classes, and wrapper classes. These concepts are essential for understanding how data is stored, processed, and managed efficiently in Java applications. As student's progress further, it becomes necessary to consolidate this knowledge and prepare for more advanced programming constructs that emphasize robustness, flexibility, and real-world applicability. Unit 14 is designed to strengthen learners' conceptual continuity by reinforcing disciplined programming practices and structured problem-solving approaches. This unit encourages students to reflect on previously learned concepts and recognize their relevance in larger program design. Programming in Java is not limited to writing syntactically correct code; it also requires logical reasoning, correct use of language features, and awareness of best practices.

The structure of this unit follows the self-learning material (SLM) approach, where learners are guided through clearly defined objectives, summarized learning outcomes, and supporting academic tools such as glossary, exercises, and suggested readings. These components help students revise key ideas, evaluate their understanding, and independently explore additional learning resources. By the end of this unit, learners will be better prepared to

NOTES

understand and apply upcoming Java concepts with confidence. Unit 14 therefore acts as an important academic link that supports smooth progression from foundational Java programming knowledge to more advanced topics covered later in the syllabus.

14.2. OBJECTIVES

After completing this unit, the learner will be able to:

- Recall relevant Java programming concepts studied in previous units
- Identify key ideas and terminology associated with this unit
- Explain the importance of structured and systematic learning in Java
- Apply learned concepts to basic programming situations
- Analyze program logic and execution flow
- Develop disciplined problem-solving skills using Java programming constructs

14.3. DEFINING CLASSES AND OBJECTS

Object-Oriented Programming (OOP) is the foundation of Java, and the concepts of classes and objects form its core. In procedural programming, data and functions are treated separately, which makes large programs difficult to manage and maintain. Java overcomes this limitation by organizing programs around objects, which combine data and the methods that operate on that data into a single unit.

A class is a blueprint or template that defines the structure and behavior of objects. It specifies what data an object will hold and what actions it can perform. An object, on the other hand, is a real-world instance of a class. While a class represents a logical concept, objects represent actual entities created in memory during program execution. For example, consider a class named Student. The class defines properties such as roll number, name, and marks, and methods such as calculating percentage or displaying details. Each individual student such as Ravi or Anita is an object of the student class. Thus, one class can be used to create many objects, each with its own data values. Defining classes and objects allows Java programs to become more modular, reusable, secure, and easier to understand. It also supports key OOP principles such as encapsulation, abstraction, inheritance, and polymorphism, which are essential for developing real-world software systems. Understanding how to define classes and create objects is therefore a fundamental step toward mastering Java programming.

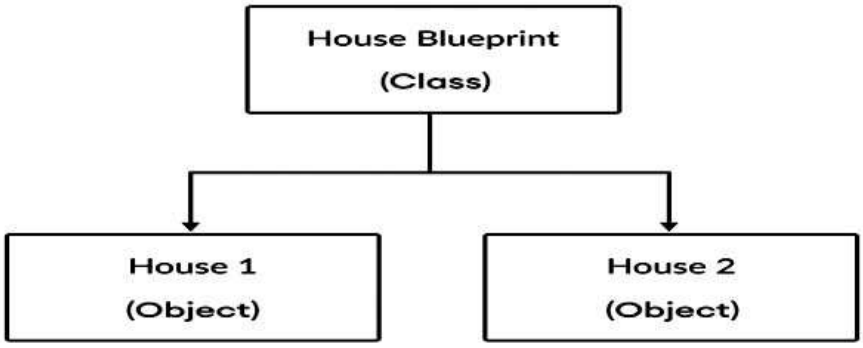


Fig. 14.1 : Defining Classes And Objects.

14.2.1. Class Definition

A class definition in Java specifies the variables (data members) and methods (member functions) that describe the behavior of objects created from that class. A class does not occupy memory by itself; memory is allocated only when objects of the class are created.

Syntax of Class Definition

```
class ClassName
{
    // data members
    // member methods
}
```

Example

```
class Student {
    int rollNo;
    String name;

    void display() {
        System.out.println(rollNo + " " + name);
    }
}
```

In this example :

- Student is the class name
- rollNo and name are data members
- display() is a member method

Characteristics of a Class

- It is a user-defined data type
- It acts as a blueprint for objects
- It supports data hiding using access specifiers
- It promotes code reusability

Classes can also include constructors, static members, and nested classes. A well-designed class improves program clarity and reduces

NOTES

complexity. Class definition is an essential step in Java programming, as it determines how objects will behave and interact within a program.

14.2.2. Object Creation

An object is an instance of a class. When an object is created, memory is allocated for its data members, and the object becomes capable of using the methods defined in the class.

Syntax for Object Creation

```
ClassName objectName = new ClassName();
```

Example

```
Student s1 = new Student();
```

Here:

- Student is the class
- s1 is the object reference
- new keyword allocates memory

Accessing Object Members

```
s1.rollNo = 101;
s1.name = "Ravi";
s1.display();
```

Each object maintains its own copy of data members, but all objects share the same method definitions.

Multiple Objects Example

```
Student s2 = new Student();
Student s3 = new Student();
```

All objects (s1, s2, s3) are created from the same class but store different values. Object creation enables Java programs to represent real-world entities and supports dynamic memory management, making applications more flexible and scalable.

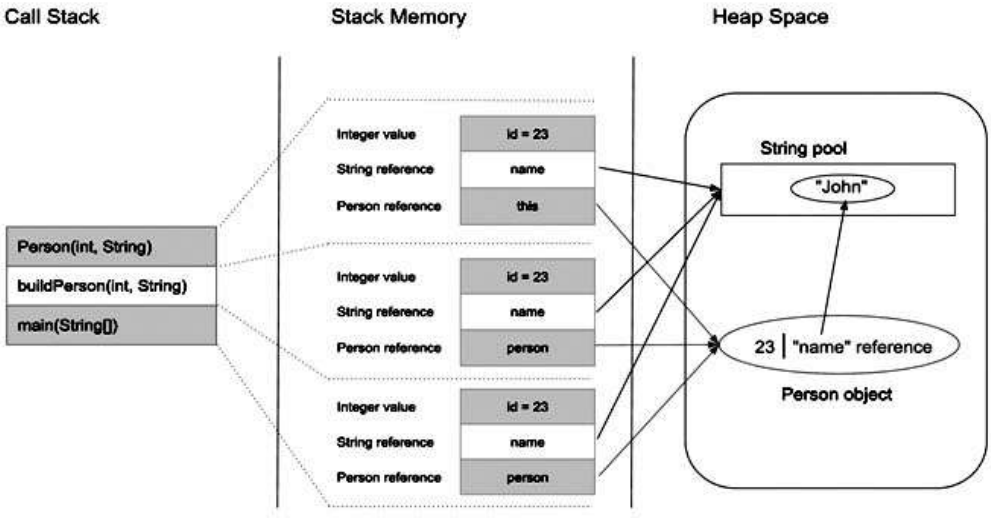


Fig. 14.2 : Object Creation.

14.4. ACCESSING CLASS MEMBERS

In Java, once a class is defined and objects are created, the next important step is accessing class members. Class members are the components that define the state and behavior of an object. They include data members (variables)and member methods (functions) declared inside a class. Accessing these members correctly is essential for effective object interaction, data manipulation, and implementation of program logic.

Java follows the principles of object-oriented programming, where data and methods are encapsulated within a class. To protect data from unauthorized access and accidental modification, Java provides access control mechanisms through access specifiers. These access rules determine which parts of a program can access specific class members. By properly controlling access, Java ensures data security, modularity, and maintainability of programs. Accessing class members usually involves using an object reference along with the dot (.) operator. In some cases, class members can also be accessed directly using the class name, especially when they are declared as static. Understanding how and when to access class members helps students write correct, secure, and well-structured Java programs. In real-world applications, accessing class members allows objects to exchange information, perform operations, and represent real-life entities such as students, employees, bank accounts, and products. Therefore, mastering this concept is a fundamental requirement for developing robust Java applications.

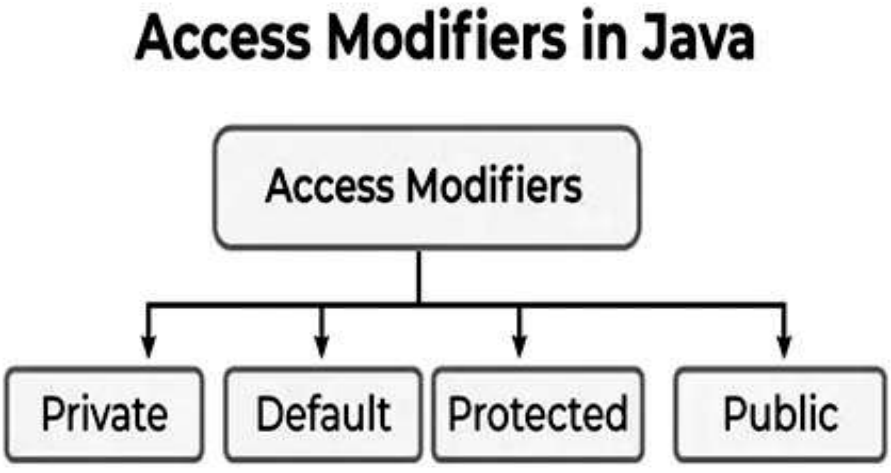


Fig. 14.3 : Access Modifiers in Java.

Access Control in Java

Java uses access specifiers to control the visibility of class members:

- **public** – Accessible from anywhere in the program
- **private** – Accessible only within the same class
- **protected** – Accessible within the same package or subclasses
- **default** (no keyword) – Accessible only within the same package

NOTES

These access levels help in implementing encapsulation, one of the core principles of object-oriented programming.

Accessing Data Members

Data members are variables declared inside a class. They represent the properties or attributes of an object. Data members can be accessed using an object reference if they are not declared as private.

Example

```
class Student
{
    int rollNo;
    String name;
    }
```

Accessing Data Members Using Object

```
Student s = new Student();
s.rollNo = 101;
s.name = "Ravi";
```

Each object maintains its own copy of data members. Changes made through one object do not affect the data of another object.

Private Data Members : When data members are declared as private, they cannot be accessed directly from outside the class. In such cases, getter and setter methods are used.

```
class Student
{
    private int rollNo;

    void setRollNo(int r)
{
        rollNo = r;
    }

    int getRollNo() {
        return rollNo;
    }
}
```

This approach ensures data security and controlled access.

Accessing Member Methods: Member methods are functions defined inside a class that operate on data members. They define the behavior of objects.

Example

class Calculator

```
{
    int add(int a, int b)
{
        return a + b;
    }
}
```

Calling Member Methods

```
Calculator c = new Calculator();
int result = c.add(10, 20);
```

Member methods can :

- Modify data members
- Perform calculations
- Display information
- Control program flow

Static Member Methods

Static methods belong to the class rather than objects and are accessed using the class name.

```
class MathUtil
{
    static int square(int x)
{
        return x * x;
    }
}
int val = MathUtil.square(5);
```

Benefits of Controlled Access

- Improves data security
- Supports encapsulation
- Makes programs easier to maintain
- Prevents accidental misuse of data
- Enhances readability and modularity

14.4.1. Data Members

Data members are variables declared inside a class. They represent the properties or attributes of an object.

Types of Data Members

- **Instance variables** – Belong to individual objects
- **Static variables** – Shared among all objects

Example

```
class Employee
{
 int id;           // instance variable
 static String company = "ABC Ltd";  // static variable
}
```

Accessing Data Members

```
Employee e1 = new Employee();
```

NOTES

```
e1.id = 101;
System.out.println(Employee.company);
```

Access Specifiers

- **public –** Accessible everywhere
- **private –** Accessible only within the class
- **protected –** Accessible within package and subclasses
- **default –** Accessible within the same package

Using appropriate access specifiers ensures data security and prevents unauthorized access to sensitive information.

14.4.2. Member Methods

In Java, member methods are functions that are defined inside a class and describe the behavior of objects created from that class. While data members represent the state or properties of an object, member methods define what the object can do. Together, data members and member methods form the core of object-oriented programming, enabling Java programs to model real-world entities effectively.

A member method operates on the data members of a class and can perform tasks such as calculations, data validation, displaying information, or modifying object state. Member methods are essential for implementing encapsulation, because they provide controlled access to data members and hide internal implementation details from the outside world. This approach improves program security, readability, and maintainability. For example, consider a student class. The data members may store roll number and marks, while member methods can calculate percentage, determine grade, or display student details. Without member methods, accessing and manipulating data directly would make programs error-prone and difficult to manage. Hence, member methods are a fundamental building block of Java classes.

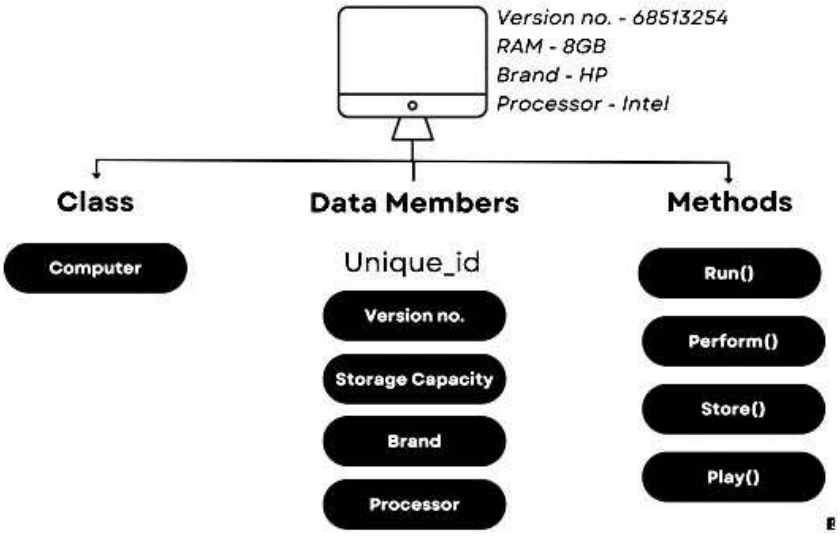


Fig. 14.4 : Member Methods.

Definition of Member Methods : A member method is a method that is declared inside a class and is associated with an object of that class. It can access both data members and other methods of the same class directly.

General Syntax

```
returnType methodName(parameters)
{
    // method body
}
```

Example

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}
```

In this example:

- add() is a member method
- int is the return type
- a and b are parameters
- The method returns the sum of two numbers

Calling Member Methods : Member methods are generally invoked using an object reference and the dot (.) operator.

Example

```
Calculator c = new Calculator();
int result = c.add(10, 20);
```

Here:

- c is the object of the class Calculator
- add() is called using the object
- The method executes and returns a value

Member methods allow objects to interact with data and perform meaningful operations.

Types of Member Methods : Member methods in Java can be broadly classified into different types based on their behavior and usage.

1. Instance Methods

Instance methods are the most common type of member methods. They are associated with objects and can access both instance variables and other instance methods.

NOTES

```
class Student {
    int marks;

    void setMarks(int m) {
        marks = m;
    }

    int getMarks() {
        return marks;
    }
}
```

Instance methods require an object to be called and are used to manipulate object-specific data.

2. Static Methods

Static methods belong to the class rather than to individual objects. They are declared using the static keyword.

```
class MathUtil {
    static int square(int x) {
        return x * x;
    }
}
Calling a Static Method
int value = MathUtil.square(5);
```

Static methods:

- Are shared by all objects
- Cannot directly access non-static data members
- Are commonly used for utility functions

3. Methods with Parameters :

Member methods may accept parameters to perform operations on input values.

```
void displayInfo(String name, int age)
{
    System.out.println(name + " " + age);
}
```

Parameters make methods flexible and reusable.

4. Methods with Return Values :

Some member methods return values to the calling program.

```
int getSum(int a, int b) {
    return a + b;
}
```

Returned values can be stored or used in expressions.

Access Specifiers and Member Methods : Member methods can be declared using access specifiers to control their visibility.

- **public –** accessible everywhere
- **private –** accessible only within the class
- **protected –** accessible within the package or subclasses
- **default –** accessible within the same package

Example

```
class Account
  {
    private double balance;

    public double getBalance() {
        return balance;
    }
}
```

Using access specifiers helps enforce encapsulation and protects sensitive data.

Advantages of Using Member Methods

- Promote **code reusability**
- Improve **data security**
- Support **modularity**
- Simplify debugging and maintenance
- Enhance program readability

Member methods allow programmers to organize logic efficiently and make programs easier to understand.

Real-World Applications of Member Methods

Member methods are used extensively in real-world Java applications, such as:

- Banking systems (deposit, withdraw methods)
- Student management systems (calculate grade, display result)
- Inventory systems (update stock, calculate total price)
- Web applications (validate input, process requests)

These examples show how member methods encapsulate behavior and make Java programs practical and scalable.

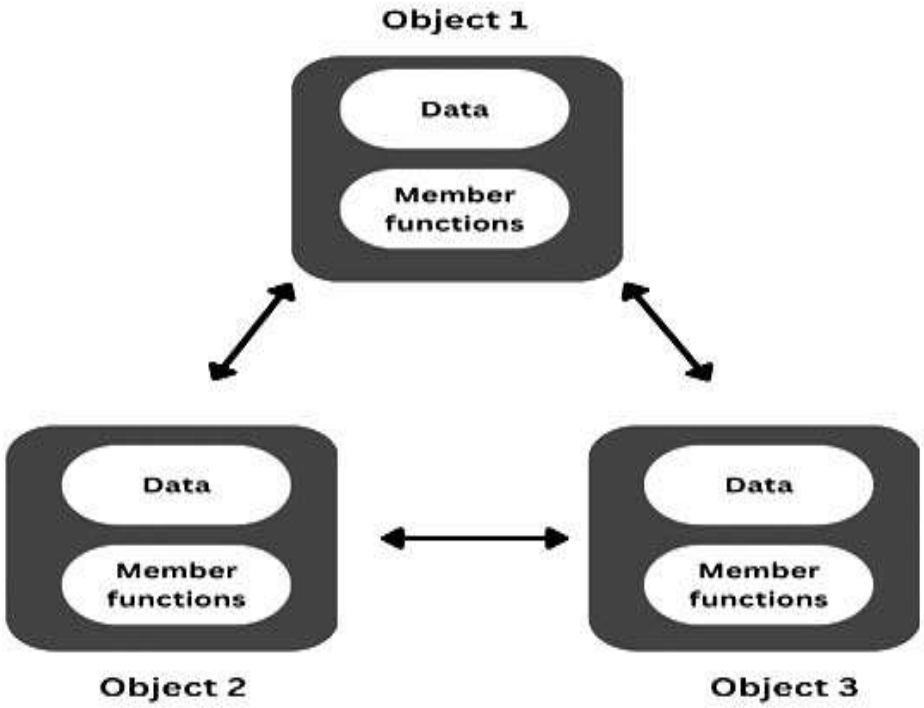


Fig. 14.5 : Member Methods.

Common Mistakes While Using Member Methods

- Forgetting to create an object before calling instance methods
- Incorrect use of static and non-static methods
- Returning wrong data types
- Poor naming conventions

Avoiding these mistakes improves program correctness and quality.

Activity 14.1. Analyzing Real-Life Objects as Classes

Objectives:

To build critical thinking by identifying class-object relationships in everyday life and strengthening conceptual clarity.

Materials Needed:

Notebook or digital document.

Activity Description:

Observe your surroundings and pick any 3 real-life objects (for example: Smartphone, Car, Bank Account).

For each object:

Define the class name

List at least 4 attributes and 3 methods

Identify one real instance (object) with actual values

Then,

compare the three classes and identify similarities or differences in their structure.

Reflection/Outcome:

Briefly answer:

How does thinking in terms of classes and objects help in organizing information?

Which object was most complex to model and why?

☑ Check Your Progress 14.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What is a class in Java? Explain its basic structure.

. . . . . .

. . . . . .

. . . . . .

2. How are objects created in Java? Explain with a brief description.

. . . . . .

. . . . . .

. . . . . .

3. What are data members and member methods? How are they accessed in a class?

. . . . . .

. . . . . .

. . . . . .

14.5. Summary

This unit reinforces continuity between earlier Java concepts and upcoming advanced topics. Clearly stated objectives guide learners toward measurable learning outcomes. Emphasis is placed on structured thinking and logical program development. The unit encourages revision of prior knowledge to strengthen conceptual clarity. Supporting elements such as glossary and exercises aid effective self-learning. Self-assessment helps learners evaluate their understanding and progress. Suggested readings promote independent and extended learning. Overall, the unit strengthens the academic base required for advanced Java programming.

14.6. GLOSSARY

- **Objective** – A clearly defined learning outcome of a unit
- **Concept Reinforcement** – Strengthening previously learned ideas
- **Syntax** – Rules that define the structure of Java statements
- **Logic** – Step-by-step reasoning used to solve problems
- **Compilation** – Conversion of source code into bytecode

- **Execution Flow** – Order in which program statements are executed
- **Runtime** – Phase during which a program runs
- **Error** – A mistake that leads to incorrect program behavior
- **Debugging** – Process of identifying and correcting errors
- **Self-Learning Material (SLM)** – Structured content designed for independent study

14.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** A class in Java is a blueprint or template used to create objects. It defines the properties (data members) and behaviors (methods) of objects. The basic structure of a class includes class name, variables (attributes), and methods (functions).
2. **Answer :** Objects are instances of a class. They are created using the `new` keyword. For example, `ClassName obj = new ClassName();`. This allocates memory and initializes the object so that it can access the class members.
3. **Answer :** Data members are variables defined inside a class that store data, while member methods are functions that define the behavior of the class. These members are accessed using an object of the class, typically with the dot (`.`) operator, such as `objectName.variable` or `objectName.method()`.

14.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3755	

14.9 References

- Balagurusamy, E. (2022). *Programming with Java*. McGraw-Hill Education.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Oracle. (2023). *Java Documentation*.
- NPTEL. (2022). *Java Programming Courses*.

14.10. Exercise

Part 1

A. Multiple Choice Question :

1. A class in Java is:
   (a) An object
   (b) A blueprint for creating objects
   (c) A method
   (d) A variable
2. Which keyword is used to create an object in Java?
   (a) create
   (b) new
   (c) class
   (d) object
3. Data members in a class are:
   (a) Functions
   (b) Variables that store data
   (c) Loops
   (d) Conditions
4. Member methods are used to:
   (a) Store values
   (b) Define behavior of a class
   (c) Declare variables
   (d) Create packages
5. Class members are accessed using:
   (a) : operator
   (b) ; operator
   (c) . (dot) operator
   (d) # operator

Answers with Explanations

1. Ans. (b)

Class Concept : A class is a blueprint used to create objects, defining their properties and behaviors.

2. Ans. (a)

Object Creation : Objects are created using the new keyword, which allocates memory for the object.

3. Ans. (b)

Data Members : Data members are variables inside a class used to store data related to objects.

4. Ans. (b)

NOTES

Member Methods : Member methods define the behavior or actions that an object can perform.

5. Ans. (c)

Accessing Members : Class members are accessed using the dot (.) operator through an object.

B. Short Answer Questions (50–100 words)

1. Explain the importance of objectives in an SLM unit.
2. Describe the role of a glossary in Java learning.
3. Why are summaries important in academic units?
4. What is the purpose of suggested readings?
5. Define debugging.

Part 2

C. Long Answer Questions (200 words)

1. Discuss how structured objectives support effective learning in Java.
2. Explain the role of exercises in self-learning materials.

Part 3

D. Long Answer Questions (300 words)

1. Explain how SLM structure supports independent learning in Java programming.
2. Discuss the importance of summaries, glossary, and exercises in reinforcing concepts.

UNIT – 15
METHODS AND CONSTRUCTORS

UNIT STRUCTURE

15.1 Introduction
15.2 Objective
15.3 Methods in Java
15.4 Constructors
15.5 Summary
15.6 Glossary
15.7 Answer to check your progress
15.8 Suggested readings - OER
15.9 Reference
15.10 Exercise

15.1. INTRODUCTION

In the preceding units, learners explored core object-oriented concepts in Java, including classes, objects, access control, and member methods. These topics established a strong foundation for understanding how Java programs are structured and how objects interact within an application. As student's progress further, it becomes essential to consolidate this knowledge while preparing to engage with more advanced programming constructs that enhance program robustness, flexibility, and scalability. Unit 15 is designed to reinforce systematic learning and disciplined programming practices. This unit emphasizes the importance of clearly defined objectives, logical organization of concepts, and continuous self-assessment. By revisiting previously learned ideas and connecting them with upcoming topics, learners can better appreciate the flow and coherence of the Java syllabus.

The unit follows a self-learning material (SLM) approach, where students are guided through explicit objectives, concise summaries, and supportive academic tools such as glossary, exercises, and suggested readings. These components are intended to help learners revise key concepts, evaluate their understanding, and independently explore additional resources for deeper insight. By the end of this unit, learners are expected to demonstrate improved confidence in applying Java concepts, analyzing program logic, and approaching problem-

NOTES

solving tasks in a structured manner. Unit 15 therefore serves as an important academic bridge that prepares students for more advanced Java topics covered in subsequent units.

15.2. OBJECTIVES

After completing this unit, the learner will be able to:

- Recall relevant Java programming concepts studied earlier
- Identify key ideas and terminology associated with this unit
- Explain the significance of structured learning components in Java
- Apply previously learned concepts in basic programming scenarios
- Analyze program logic and execution flow
- Develop systematic and disciplined problem-solving skills

15.3. METHODS IN JAVA

In Java programming, methods are fundamental building blocks that define the behavior of a class. A method is a block of code that performs a specific task and can be executed whenever it is called. Methods help in breaking large programs into smaller, manageable units, making the code easier to read, debug, test, and maintain. Without methods, programs would become lengthy, repetitive, and difficult to understand. A method in Java may accept input in the form of parameters, perform operations on those inputs, and optionally return a result to the caller. By using methods, programmers can reuse code, reduce duplication, and implement modular programming practices. For example, instead of writing the same calculation logic multiple times, a single method can be written and reused whenever needed. Methods also play a crucial role in object-oriented programming. They allow objects to interact with each other by invoking methods that operate on object data. Methods support core OOP principles such as encapsulation (by hiding implementation details) and abstraction (by exposing only necessary functionality).

In Java, methods can be:

- Instance methods (called using objects)
- Static methods (called using class name)
- Methods with parameters
- Methods with return values

Understanding methods is essential before learning advanced concepts such as constructors, inheritance, and polymorphism.

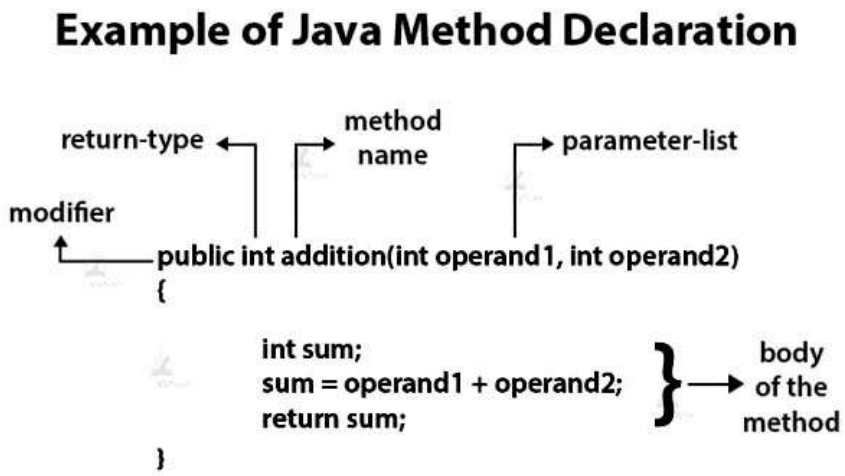


Fig. 15.1 : Java Method Declaration

15.3.1 Call by Value

Java uses call by value as its parameter-passing mechanism. In call by value, a copy of the actual value is passed to the method. Any changes made to the parameter inside the method do not affect the original variable used in the calling method.

Concept Explanation

When a variable is passed to a method:

- Java creates a copy of that variable's value
- The method works only on the copied value
- Original value remains unchanged

Example

```
class Test {
    void change(int x) {
        x = x + 10;
    }

    public static void main(String[] args) {
        int a = 5;
        Test t = new Test();
        t.change(a);
        System.out.println(a);
    }
}
```

Output : 5

Here, even though x is modified inside the method, the original variable a remains unchanged.

Advantages of Call by Value

- Prevents accidental modification of original data
- Improves program safety
- Makes code predictable

NOTES

Java strictly follows call by value, even when objects are passed (the reference itself is passed by value).

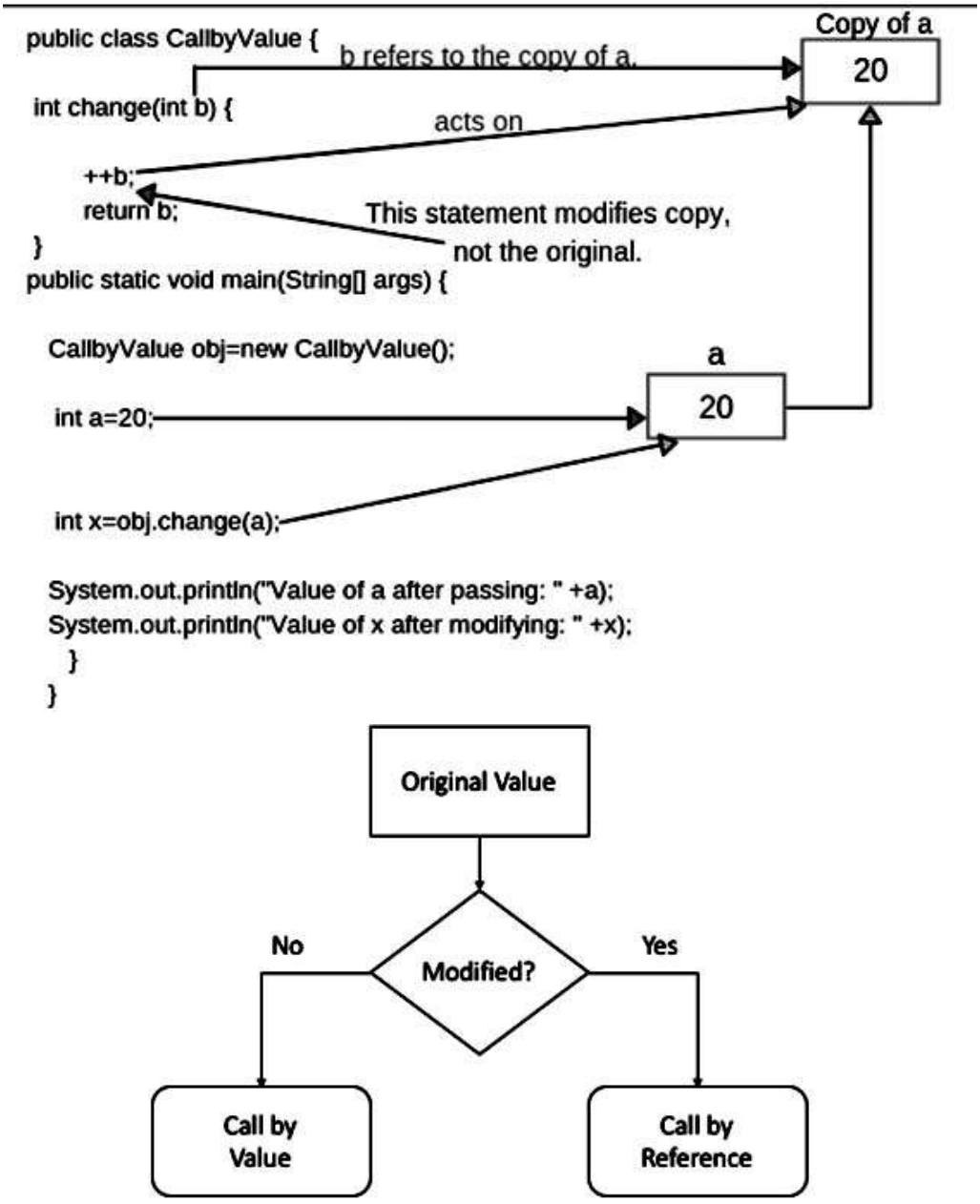


Fig. 15.2 : Call by Value mechanism.

15.3.2. Call by Reference (Java Perspective)

Strictly speaking, Java does not support call by reference in the traditional sense. However, when objects are passed to methods, Java passes a copy of the reference, which allows the method to modify the object's internal state. This behavior is often informally referred to as "call by reference."

Explanation

- The reference (address) of the object is copied
- Both original and copied reference point to the same object

- Changes to object data inside the method are reflected outside

Example

```
class Sample {
    int data = 50;

    void change(Sample s) {
        s.data = s.data + 100;
    }

    public static void main(String[] args) {
        Sample obj = new Sample();
        obj.change(obj);
        System.out.println(obj.data);
    }
}
```

Output : 150

Here, the object's data is changed because both references point to the same object.

Key Difference from True Call by Reference

- Reassigning the object reference inside the method does not affect the original reference
- Only object contents can be modified

This concept is important for understanding object manipulation and memory behavior in Java.

15.3.3. Recursion

Recursion is a programming technique in which a method calls itself to solve a problem. Recursive solutions are particularly useful for problems that can be divided into smaller subproblems of the same type, such as factorial calculation, Fibonacci series, and tree traversal.

Structure of a Recursive Method

A recursive method must have:

1. **Base condition –** stops recursion
2. **Recursive call –** method calls itself

Example : Factorial

```
int factorial(int n) {
    if(n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

NOTES

How Recursion Works

- Each method call is stored in the call stack
- Execution continues until base condition is reached
- Stack unwinds and returns values step by step

Advantages

- Simplifies complex problems
- Reduces code length
- Improves logical clarity

Limitations

- Higher memory usage
- Risk of stack overflow
- Slower than loops in some cases

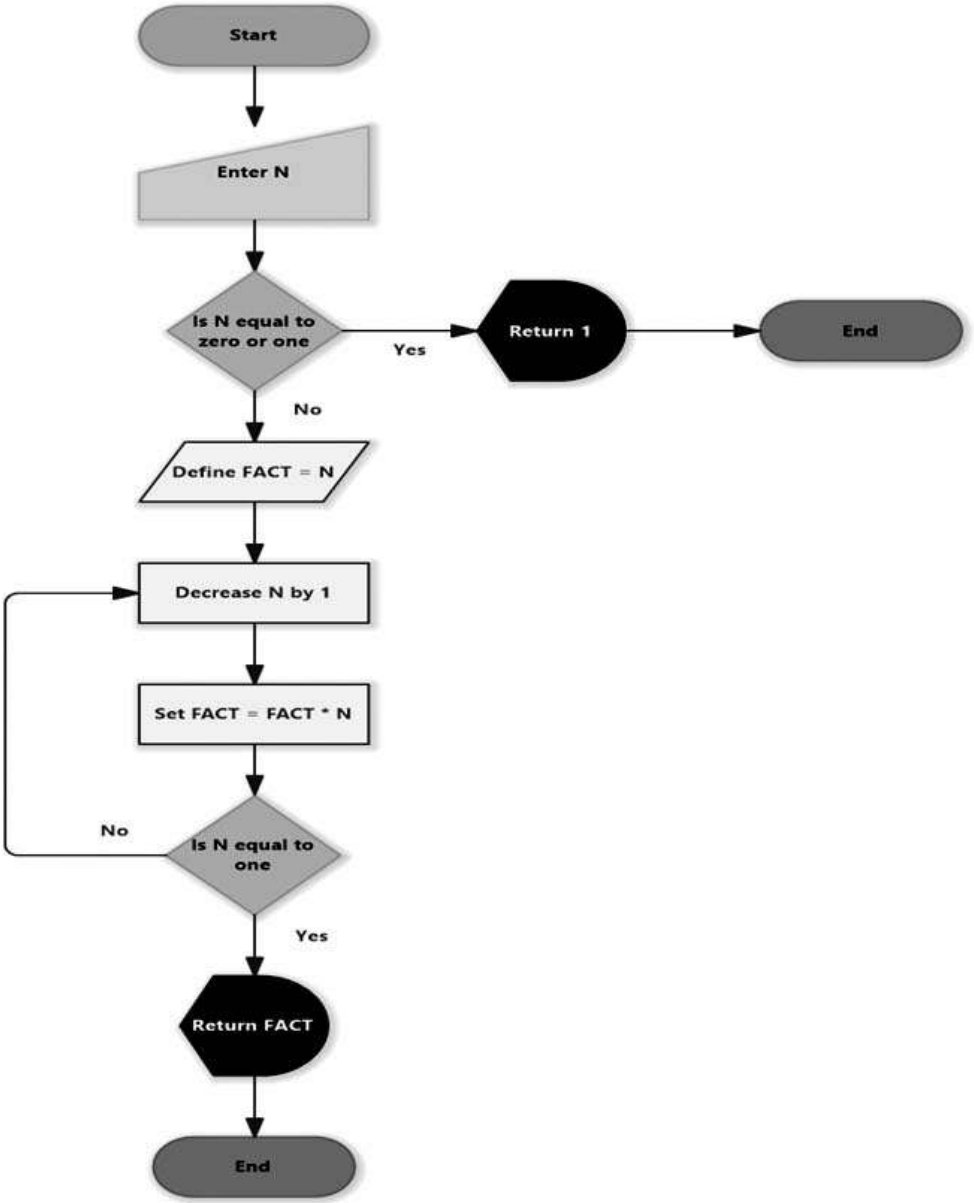


Fig. 15.3 : Limitations.

15.4. CONSTRUCTORS

In Java, a constructor is a special member of a class that is used to initialize objects at the time of their creation. When an object is created using the new keyword, the constructor of the class is automatically invoked. The main purpose of a constructor is to assign initial values to the data members of a class so that the object starts its life in a valid and consistent state.

Constructors play a vital role in object-oriented programming, as they ensure that objects are properly initialized before they are used. Without constructors, programmers would have to manually assign values to object variables after object creation, which could lead to errors, incomplete initialization, or misuse of objects. By using constructors, Java enforces a disciplined approach to object creation.

A constructor has the same name as the class and does not have a return type, not even void. Unlike ordinary methods, constructors are called automatically and cannot be invoked explicitly. Every class in Java has at least one constructor. If no constructor is defined by the programmer, Java provides a default constructor that initializes data members with default values. Understanding constructors is essential for writing robust Java programs, as they are closely associated with concepts such as encapsulation, object initialization, and memory management.

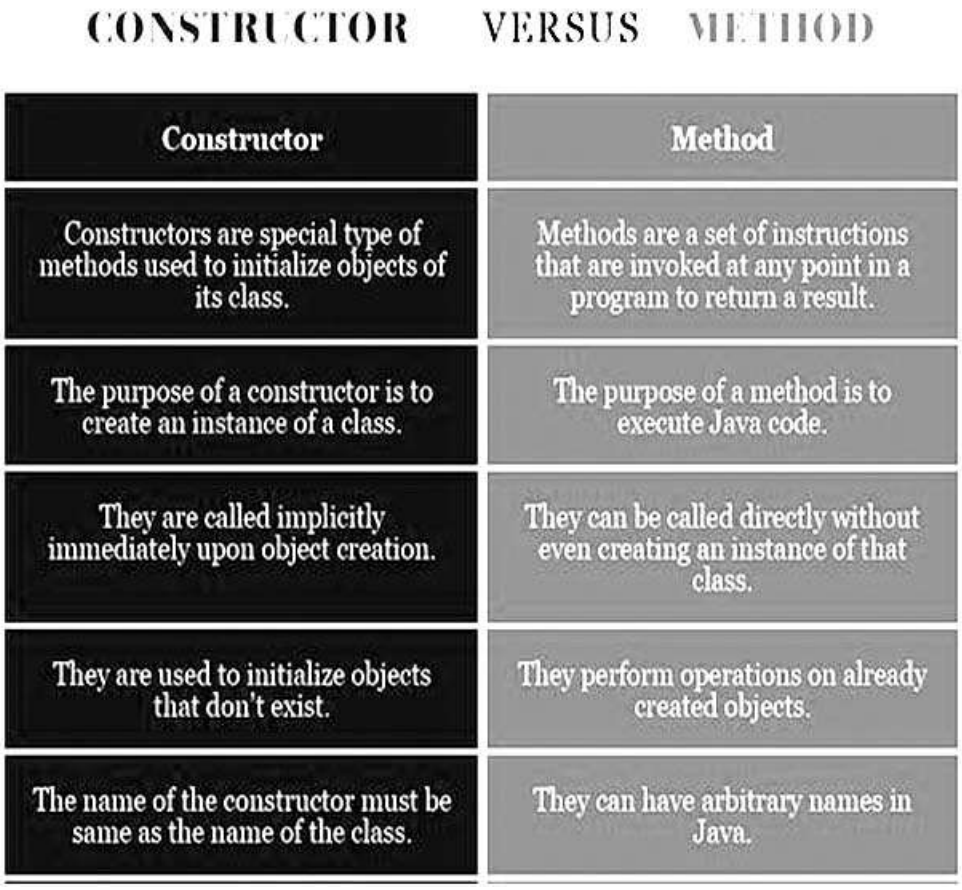


Fig. 15.4 : Constructors.

NOTES

Characteristics of Constructors

Constructors in Java have the following important characteristics:

- The constructor's name must be the same as the class name
- Constructors do not return any value
- They are invoked automatically when an object is created
- Constructors can be overloaded
- They can have access specifiers such as public, private, or protected
- They are executed only once per object creation

These characteristics make constructors distinct from ordinary methods and emphasize their role in object initialization.

Types of Constructors

Java mainly supports two types of constructors:

1. Default Constructor

A **default constructor** is a constructor with no parameters. If a class does not explicitly define any constructor, Java automatically provides a default constructor.

Example:

```
class Sample {
    int x;
}
```

Here, Java internally provides a default constructor that initializes x to 0.

If the programmer defines a constructor explicitly, Java does not provide a default constructor automatically.

2. Parameterized Constructor

A parameterized constructor accepts parameters and uses them to initialize data members.

Example:

```
class Student {
    int rollNo;

    Student(int r) {
        rollNo = r;
    }
}
```

Object creation:

```
Student s = new Student(101);
```

Parameterized constructors allow objects to be initialized with specific values, making programs more flexible and meaningful.

Constructor Execution and Object Initialization

When an object is created using the new keyword, the following steps occur:

1. Memory is allocated for the object in heap memory
2. Data members are initialized with default values
3. The constructor is executed
4. The object reference is returned

This sequence ensures that every object is fully initialized before it is used in the program.

Advantages of Using Constructors

- Ensure proper initialization of objects
- Reduce chances of errors due to uninitialized variables
- Improve program readability and reliability
- Support encapsulation by controlling object creation
- Allow flexible initialization using parameters

Constructors are widely used in real-world Java applications such as student management systems, banking software, and enterprise-level systems where object consistency is critical.

Difference Between Constructor and Method

Constructor	Method
Same name as class	Can have any valid name
No return type	Must have a return type
Called automatically	Called explicitly
Used for initialization	Used to perform operations

This comparison highlights why constructors are treated as a special component of Java classes.

Limitations of Constructors

- Constructors cannot be inherited
- They cannot be overridden
- Excessive constructor overloading can reduce readability

Despite these limitations, constructors remain a fundamental concept in Java programming.

15.4.1. Constructor Concept

The primary purpose of a constructor is to initialize object data members at the time of object creation.

NOTES

Example

```
class Student {
    int rollNo;

    Student(int r) {
        rollNo = r;
    }
}
```

Default Constructor

If no constructor is defined, Java provides a default constructor that initializes data members with default values.

Benefits of Constructors

- Automatic initialization
- Improves object consistency
- Reduces errors

Constructors improve program reliability and readability.

15.4.2. Constructor Overloading

Constructor overloading is a concept where a class has multiple constructors with different parameter lists. It allows objects to be created in different ways.

Example

```
class Rectangle {
    int length, breadth;

    Rectangle() {
        length = breadth = 0;
    }

  Rectangle(int l, int b) {
        length = l;
        breadth = b;
    }
}
```

Advantages

- Flexible object creation
- Improves code usability
- Supports polymorphism

Constructor overloading is widely used in real-world Java applications to provide multiple initialization options.

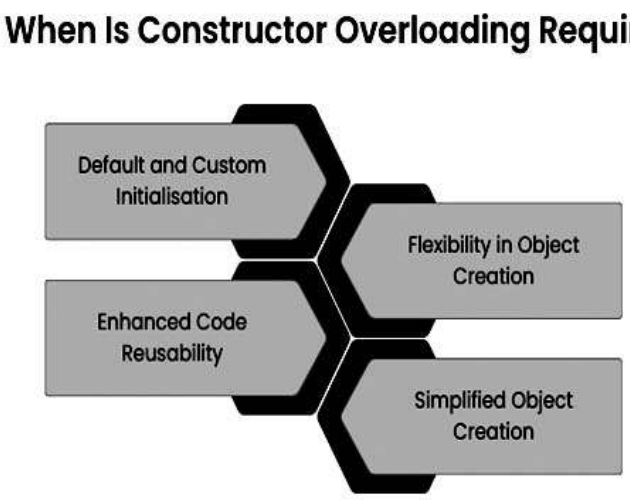


Fig. 15.5 : Constructor Overloading.

Activity 15.1.

Write a class with multiple methods and constructors. Show how constructors initialize objects.

☑ Check Your Progress 15.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What is a method in Java? Explain call by value and call by reference.

. . . . . .

. . . . . .

. . . . . .

2. What is recursion? Explain how it works in Java.

. . . . . .

. . . . . .

. . . . . .

3. What is a constructor? Explain constructor overloading.

. . . . . .

. . . . . .

. . . . . .

15.5. SUMMARY

- This unit reinforces continuity between earlier Java concepts and forthcoming advanced topics
- Clearly stated objectives guide learners toward measurable learning outcomes
- Emphasis is placed on logical thinking and structured program development

- The unit supports revision of prior knowledge to strengthen conceptual clarity
- Academic tools such as glossary and exercises enhance self-learning.
- Self-assessment enables learners to evaluate understanding and progress
- Suggested readings promote independent and extended learning
- Overall, the unit strengthens the academic foundation required for advanced Java programming

15.6. GLOSSARY

- **Objective –** A clearly defined and measurable learning outcome
- **Concept Reinforcement –** Strengthening previously learned ideas
- **Syntax –** Rules that govern the structure of Java statements
- **Logic –** Step-by-step reasoning used to solve programming problems
- **Compilation –** Process of converting source code into bytecode
- **Execution Flow –** Order in which program instructions are executed
- **Runtime –** Phase during which a program is running
- **Error –** A mistake that causes incorrect program behavior
- **Debugging –** Process of identifying and fixing errors in programs
- **Self-Learning Material (SLM) –** Structured content designed for independent study

15.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** A method in Java is a block of code that performs a specific task and is executed when called. Java primarily uses call by value, where a copy of the variable is passed to the method, so changes do not affect the original value. In contrast, call by reference (conceptually for objects) passes the reference of an object, allowing changes to affect the original object.
2. **Answer :** Recursion is a process where a method calls itself repeatedly until a base condition is met. It is useful for solving problems like factorial **calculation**, tree traversal, and searching algorithms. Each recursive call reduces the problem size until it reaches the base case, stopping further calls.
3. **Answer :** A constructor is a special method used to initialize objects. It has the same **name** as the class and does not have a return type. Constructor overloading means defining multiple constructors in a class with different parameter lists, allowing objects to be initialized in different ways.

15.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3756	

15.9 REFERENCES

- Balagurusamy, E. (2022). *Programming with Java*. McGraw-Hill Education.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Oracle. (2023). *Java Documentation*.
- NPTEL. (2022). *Java Programming Courses*.

15.10. EXERCISE

Part 1

A. Multiple Choice Question

1. A method in Java is:
   (a) A variable
   (b) A block of code that performs a task
   (c) A class
   (d) A package
2. Java primarily uses:
   (a) Call by reference
   (b) Call by value
   (c) Both equally
   (d) None
3. Recursion is a process where:
   (a) A method calls another method
   (b) A method calls itself
   (c) A loop repeats infinitely
   (d) A variable changes value
4. A constructor is:
   (a) A normal method with return type
   (b) A special method used to initialize objects
   (c) A loop structure
   (d) A data member
5. Constructor overloading means:

NOTES

(a) Using one constructor only

(b) Defining multiple constructors with same name but different parameters

(c) Deleting constructors

(d) Calling constructors multiple times

Answers with Explanations

1. Ans. (b)

Method Concept : A method is a block of code that performs a specific task and is executed when called.

2. Ans. (b)

Call by Value : Java uses call by value, where a copy of the variable is passed to methods.

3. Ans. (b)

Recursion : Recursion occurs when a method calls itself repeatedly until a base condition is met.

4. Ans. (b)

Constructor Concept : A constructor is a special method used to initialize objects when they are created.

5. Ans. (b)

Constructor Overloading : Constructor overloading allows multiple constructors in a class with different parameter lists.

B. Short Answer Questions (50–100 words)

1. Explain the importance of objectives in an SLM unit.
2. Describe the role of a glossary in learning Java.
3. Why are summaries important in academic units?
4. What is the purpose of suggested readings?
5. Define debugging.

Part 2

C. Long Answer Questions (200 words)

1. Discuss how structured objectives support effective learning in Java.
2. Explain the role of exercises in self-learning materials.

Part 3

D. Long Answer Questions (300 words)

1. Explain how SLM structure supports independent learning in Java programming.
2. Discuss the importance of summaries, glossary, and exercises in reinforcing concepts.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | BCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC | A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

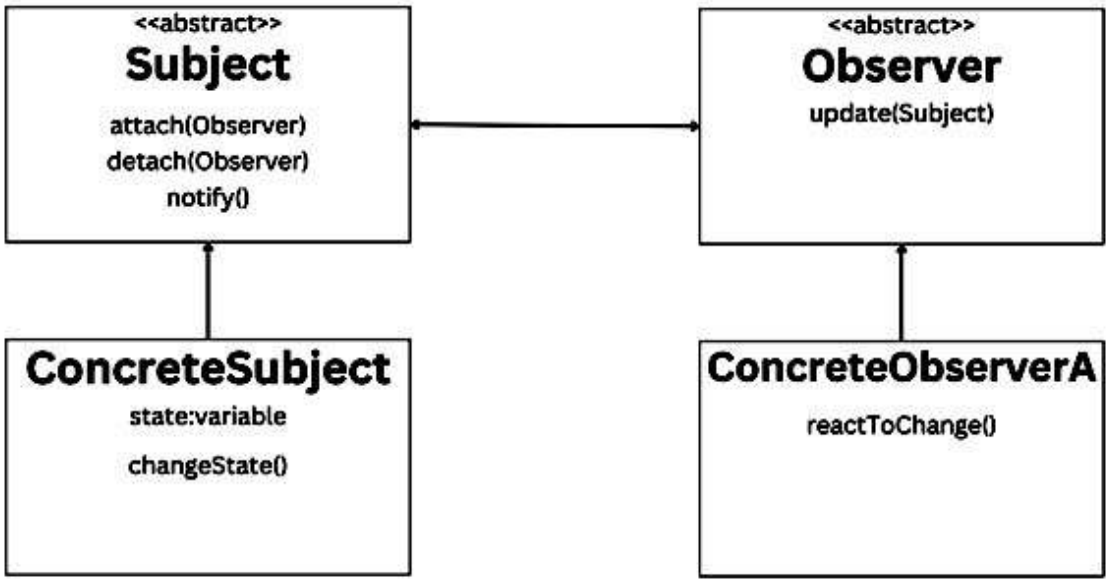
_____

_____

_____

***

Block VI	ADVANCED OBJECT-ORIENTED AND MULTITASKING CONCEPTS	225 – 292
Unit 16	Advanced Class Concepts	227
Unit 17	Inheritance	239
Unit 18	Polymorphism and Abstraction	251
Unit 19	Interfaces, Packages and Exceptions	263
Unit 20	Multithreading, Applets, I/O and Graphics	277



This block introduces learners to advanced features of object-oriented programming in Java, building upon previously acquired foundational knowledge. It explores deeper concepts such as advanced class design, inheritance, polymorphism, and abstraction, which are essential for developing flexible and reusable code. The block further extends to interfaces, packages, and exception handling, enabling learners to create well-structured and error-resilient applications. Additionally, it covers multithreading, applets, input/output operations, and graphics, providing insight into concurrent programming and interactive application development. Overall, this block equips learners with the conceptual understanding and practical skills required to design efficient, scalable, and real-world Java applications.

UNIT – 16
ADVANCED CLASS CONCEPTS

UNIT STRUCTURE

16.1 Introduction
16.2 Objective
16.3 Overloading and Static Members
16.4 Memory Management
16.5 Summary
16.6 Glossary
16.7 Answer to check your progress
16.8 Suggested readings - OER
16.9 Reference
16.10 Exercise

16.1. INTRODUCTION

In the previous units, learners have developed a solid understanding of core object-oriented programming concepts in Java, including methods, constructors, and object creation. These concepts are essential for building structured, reusable, and maintainable programs. As students move forward in the Java syllabus, it becomes important to reinforce these foundations while gradually preparing for more advanced programming constructs that support real-world application development.

Unit 16 is designed to strengthen conceptual clarity and promote systematic learning. This unit encourages learners to review previously acquired knowledge and understand how individual programming components fit together in larger program structures. Java programming requires not only correct syntax but also logical thinking, disciplined coding practices, and awareness of best practices. This unit supports learners in developing these essential skills. The self-learning material (SLM) approach adopted in this unit provides clear objectives, concise summaries, and structured academic support through glossary, exercises, and suggested readings. These components help learners assess their understanding, revise key ideas, and explore additional learning resources independently. By the end of this unit, learners are expected to demonstrate improved confidence in applying Java concepts, analyzing program behavior, and approaching programming problems in a structured and logical

NOTES

manner. Unit 16 thus acts as an academic bridge that prepares students for more complex Java topics covered in subsequent units.

16.2. OBJECTIVES

After completing this unit, the learner will be able to:

- **Recall** relevant Java programming concepts studied in earlier units
- **Identify** important ideas and terminology related to this unit
- **Explain** the role of structured learning in Java programming
- **Apply** previously learned concepts to simple programming scenarios
- **Analyze** program logic and execution flow
- **Develop** disciplined and systematic problem-solving skills

16.3. OVERLOADING AND STATIC MEMBERS

Java provides powerful features that improve code flexibility, reusability, and efficiency. Two such important features are overloading and static members. Overloading allows programmers to define multiple methods with the same name but different parameter lists, while static members belong to the class rather than to individual objects. Together, these concepts help in writing cleaner, more organized, and memory-efficient programs. Overloading is closely related to compile-time polymorphism, where the method to be executed is determined during compilation. Static members, on the other hand, are associated with the class itself and are shared among all objects. Understanding both concepts is essential for designing scalable and optimized Java applications.

16.3.1. Method Overloading

Method overloading is a feature in Java that allows a class to have multiple methods with the same name, provided their parameter lists are different. The difference may be in:

- Number of parameters
- Type of parameters
- Order of parameters

The return type alone cannot differentiate overloaded methods.

Method overloading improves readability and usability of programs by allowing the same logical operation to be represented by a single method name.

Example of Method Overloading

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
```

```
    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

Here, the method `add()` is overloaded to perform addition for different types and numbers of arguments.

How Java Resolves Overloaded Methods

- Java uses method signature (method name + parameter list)
- Resolution happens at compile time
- This is known as static binding

Advantages of Method Overloading

- Improves code readability
- Reduces method naming complexity
- Supports polymorphism
- Enhances program flexibility

Limitations

- Excessive overloading may reduce clarity
- Confusing parameter lists can cause errors

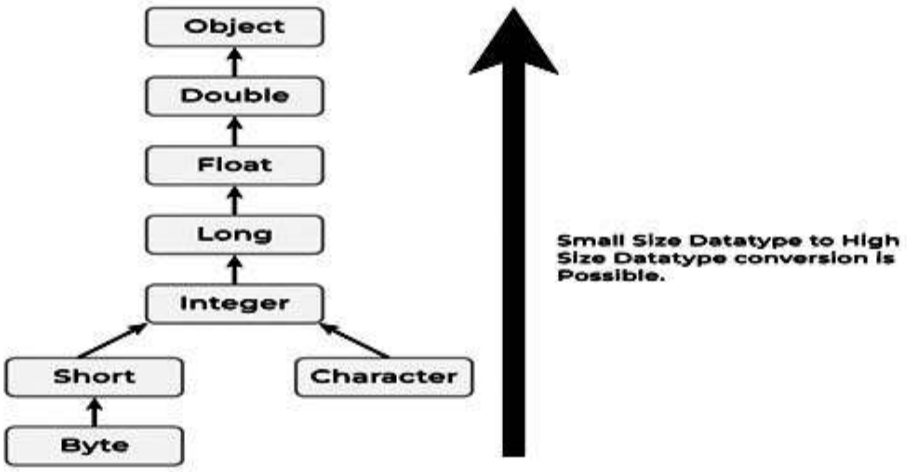


Fig. 16.1 : Method Overloading.

16.3.2. Static Variables and Methods

In Java, the keyword static is used to define class-level members. Static members belong to the class itself, not to individual objects. This means only one copy of a static variable or method exists, regardless of how many objects are created.

Static Variables

NOTES

A static variable is shared among all objects of a class.

```
class Student {
    int rollNo;
    static String college = "ABC University";
}
```

Here, college is a static variable and is common to all students.

Accessing Static Variables

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

Static Methods

Static methods are also class-level methods and can be called using the class name.

```
class MathUtil {
    static int square(int x) {
        return x * x;
    }
}
int result = MathUtil.square(5);
```

Key Characteristics

- Static methods cannot access non-static members directly
- Static members are initialized before objects are created
- **Commonly used for utility methods**

Advantages

- Memory efficient
- Faster access
- Ideal for shared data and utilities

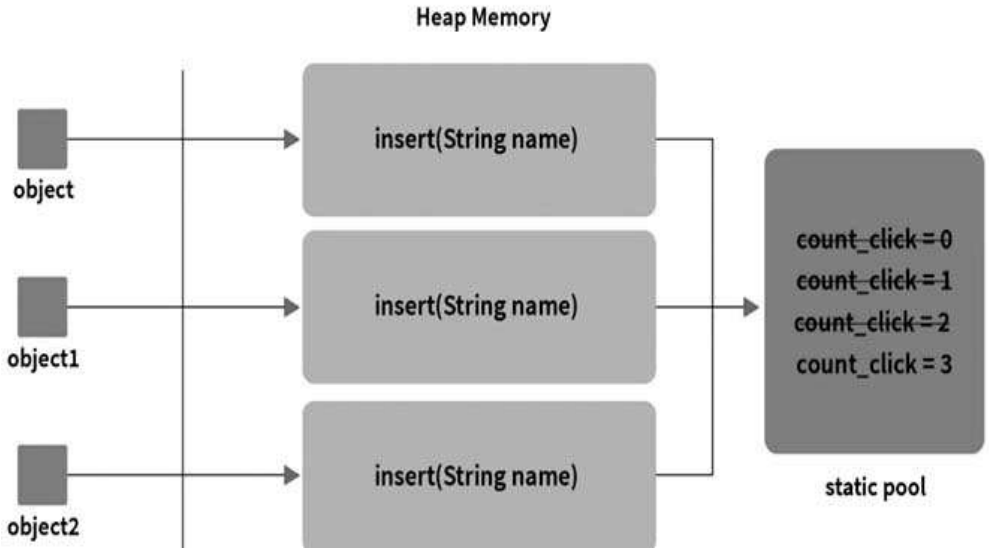


Fig. 16.2 : Static Variables and Methods.

16.4. MEMORY MANAGEMENT

Memory management in Java refers to the process of allocating and deallocating memory automatically. Unlike low-level languages such as C and C++, Java handles memory management internally, reducing programmer burden and minimizing memory-related errors.

Java memory is mainly divided into:

- **Heap memory –** stores objects
- **Stack memory –** stores method calls and local variables

The Java Virtual Machine (JVM) is responsible for memory allocation and deallocation, making Java a safer and more reliable language. Efficient memory management improves program performance, stability, and scalability.

16.4.1. Garbage Collection

Garbage Collection (GC) is an automatic memory management process in Java that identifies and removes unused objects from heap memory. An object becomes eligible for garbage collection when there are no references pointing to it.

Why Garbage Collection is Needed

- Prevents memory leaks
- Frees unused memory
- Improves application performance

Example

```
Student s1 = new Student();
s1 = null;   // object eligible for garbage collection
```

How Garbage Collection Works

- JVM identifies unreachable objects
- Frees heap memory
- Runs automatically in background

Ways an Object Becomes Eligible

- Reference set to null
- Object reference reassigned
- Object created inside a method

Advantages

- Automatic memory cleanup
- Reduces programmer errors
- Enhances program reliability

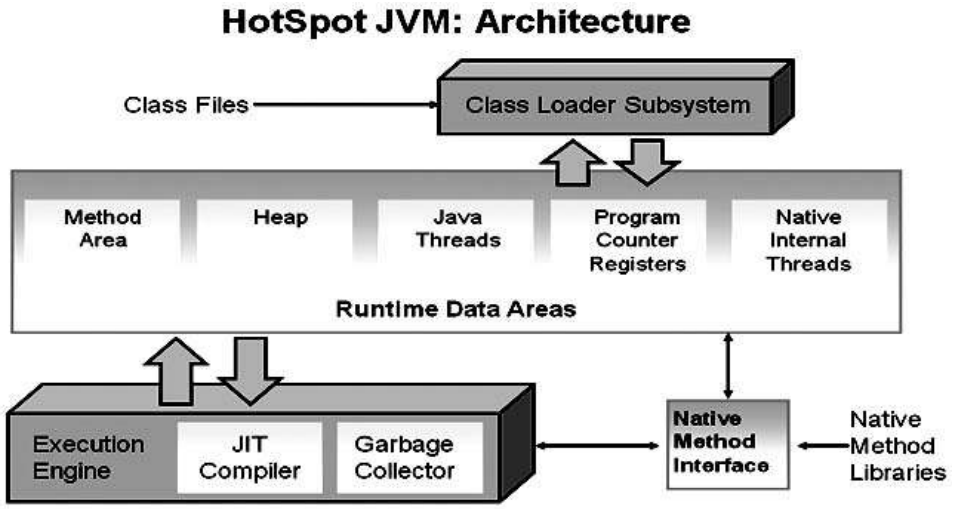


Fig. 16.3 : HotSpot JVM: Architecture.

16.4.2. Finalize() Method

In Java, memory management is handled automatically by the Java Virtual Machine (JVM) through a process known as garbage collection. While this automatic mechanism significantly reduces the burden on programmers, there are situations where an object may be holding external resources such as file handles, database connections, or network sockets. Before such an object is removed from memory, these resources may need to be released properly. To support this requirement, Java provides a special method known as the finalize() method. The finalize() method is defined in the Object class and is invoked by the garbage collector just before an object is destroyed. Its primary purpose is to allow cleanup operations to be performed on an object before it is permanently removed from memory. Unlike constructors, which are called automatically when an object is created, the finalize()method is called automatically by the garbage collector when the object becomes eligible for garbage collection. It is important to note that programmers do not explicitly call the finalize() method in normal execution. Instead, it is called by the JVM at an unpredictable time, depending on memory availability and garbage collection cycles. This uncertainty has significant implications for how and when the finalize() method should be used.

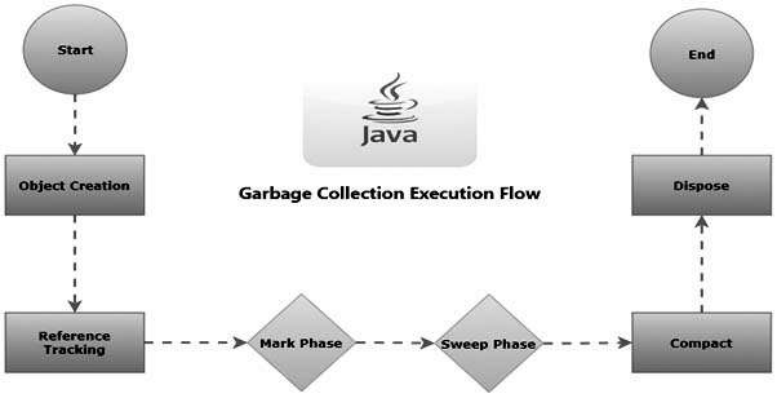


Fig. 16.4 : finalize() Method.

Definition and Syntax of finalize() Method

The finalize() method is declared in the Object class with the following syntax :

```
protected void finalize() throws Throwable {
    // cleanup code
}
```

Key points about the syntax:

- The method is declared as protected
- It does not return any value
- It can throw a Throwable
- It is inherited by all Java classes

A class can override the finalize() method to define custom cleanup behavior.

Working of finalize() Method

The life cycle of an object in relation to the finalize() method involves the following steps:

1. An object is created and used during program execution
2. The object becomes unreachable (no active references)
3. The object becomes eligible for garbage collection
4. The garbage collector may invoke finalize()
5. After finalization, the object is removed from memory

The execution of finalize() depends entirely on the JVM and garbage collector. There is no guarantee that it will be executed immediately or even at all before the program terminates.

Example of finalize() Method

```
class Demo {
    protected void finalize() {
       System.out.println("Object is being garbage collected");
    }

    public static void main(String[] args) {
        Demo d = new Demo();
        d = null;
        System.gc();
    }
}
```

In this example:

- The object becomes eligible for garbage collection
- The finalize() method may be called
- Output is not **guaranteed,** as execution depends on JVM behavior

NOTES

This example demonstrates why relying on finalize() for critical operations is unsafe.

Uses of finalize() Method

Historically, the finalize() method was used for:

- Releasing file resources
- Closing database connections
- Cleaning up native resources
- Performing last-minute object cleanup

However, due to unpredictable execution, its use has declined significantly in modern Java applications.

Limitations of finalize() Method

The finalize() method has several serious limitations:

1. **No Execution Guarantee**

   The JVM does not guarantee that finalize() will be executed before program termination.

2. **Unpredictable Timing**

   It is not possible to determine when the garbage collector will call finalize().

3. **Performance Overhead**

   Objects with finalize() take longer to be garbage collected.

4. **Security Risks**

   Improper use can lead to resource leaks and vulnerabilities.

5. **Deprecated in Modern Java**

   The finalize() method has been deprecated in newer versions of Java.

Alternatives to finalize() Method

Due to its limitations, Java recommends safer alternatives:

- try-with-resources for automatic resource management
- Explicit cleanup methods like close()
- Use of AutoCloseable interface
- Proper exception handling

These approaches provide deterministic resource management, unlike finalize().

Comparison: finalize() vs Explicit Cleanup

finalize() Method	Explicit Cleanup
Called by JVM	Called by programmer
Unpredictable	Predictable
Deprecated	Recommended
Risky	Safe

This comparison clearly shows why modern Java discourages the use of finalize().

Academic Importance of finalize() Method

From an academic perspective, the finalize() method is important because:

- It explains how Java manages object destruction
- It introduces limitations of automatic cleanup
- It helps students understand JVM internals
- It provides historical context in Java evolution

Even though it is deprecated, understanding finalize() helps learners appreciate why better memory management practices are used today.

Activity 16.1.

Experiment with concepts like static members and nested classes. Write examples demonstrating their use.

☑ Check Your Progress 16.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What is method overloading? Explain its importance in Java.

   . . . . . .

   . . . . . .

   . . . . . .

2. What are static variables and static methods? How are they used?

   . . . . . .

   . . . . . .

   . . . . . .

3. Explain garbage collection and the role of the finalize() method in memory management.

   . . . . . .

   . . . . . .

   . . . . . .

16.5. SUMMARY

This unit reinforces continuity between previously learned Java concepts and upcoming advanced topics. Clearly defined objectives guide learners toward measurable learning outcomes. Emphasis is placed on logical reasoning and structured program development.

The unit supports revision of foundational concepts to strengthen understanding. Academic tools such as glossary and exercises

NOTES

enhance self-learning. Self-assessment helps learners evaluate their progress and comprehension. Suggested readings encourage independent and extended learning. Overall, the unit strengthens the academic base required for advanced Java programming

16.6. GLOSSARY

- **Objective –** A clearly stated and measurable learning outcome
- **Concept Reinforcement –** Strengthening previously learned programming ideas
- **Syntax –** Rules that govern the structure of Java statements
- **Logic –** Step-by-step reasoning used to solve programming problems
- **Compilation –** Process of converting Java source code into bytecode
- **Execution Flow –** Order in which program instructions are executed
- **Runtime –** Phase during which a program is running
- **Error –** A mistake that causes incorrect program behavior
- **Debugging –** Process of identifying and correcting program errors
- **Self-Learning Material (SLM) –** Structured content designed for independent study

16.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer:** Method overloading is the process of defining multiple methods in the same class with the same name but different parameter lists (type, number, or order). It improves code readability and flexibility by allowing the same method name to perform different tasks based on input.
2. **Answer:** Static variables are shared among all objects of a class, meaning only one copy exists. Static methods belong to the class rather than an instance and can be called without creating an object. They are commonly used for utility functions and shared data.
3. **Answer:** Garbage collection is an automatic memory management process in Java that removes unused objects to free memory. The finalize() method is called by the garbage collector before destroying an object, allowing cleanup operations. However, it is rarely used in modern Java as it is not reliable and has been deprecated in recent versions.

16.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com /User/UnitUrls?unitId=3757	

16.9. REFERENCES

- Balagurusamy, E. (2022). *Programming with Java*. McGraw-Hill Education.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Oracle. (2023). *Java Documentation*.
- NPTEL. (2022). *Java Programming Courses*.

16.10 EXERCISE

Part 1

A. Multiple Choice Question

1. Method overloading means:
   (a) Using different method names
   (b) Using same method name with different parameters
   (c) Using only one method
   (d) Deleting methods
2. Static variables are:
   (a) Created for each object
   (b) Shared among all objects of a class
   (c) Local to methods
   (d) Temporary variables
3. Static methods can be called:
   (a) Only with objects
   (b) Without creating an object
   (c) Only inside constructors
   (d) Only in loops
4. Garbage collection is used for:
   (a) Increasing memory
   (b) Removing unused objects from memory
   (c) Storing data permanently
   (d) Compiling programs
5. The finalize() method is:

NOTES

(a) Used to start a program

(b) Called before object destruction

(c) Used to declare variables

(d) Used for input operations

Answers with Explanations

1. Ans. (b)

Method Overloading : Method overloading allows multiple methods with the same name but different parameter lists, improving flexibility and readability.

2. Ans. (b)

Static Variables : Static variables are shared among all instances of a class and exist only once in memory.

3. Ans. (b)

Static Methods : Static methods belong to the class and can be called without creating an object.

4. Ans. (b)

Garbage Collection : Garbage collection automatically removes unused objects from memory, improving memory management.

5. Ans. (b)

finalize() Method : The finalize() method is called by the garbage collector before destroying an object to perform cleanup tasks.

B. Short Answer Questions (50–100 words)

1. Explain the importance of objectives in an SLM unit.
2. Describe the role of a glossary in Java learning.
3. Why are summaries important in academic units?
4. What is the purpose of suggested readings?
5. Define debugging.

Part 2

C. Long Answer Questions (200 words)

1. Discuss how structured objectives support effective learning in Java.
2. Explain the role of exercises in self-learning materials.

Part 3

D. Long Answer Questions (300 words)

1. Explain how SLM structure supports independent learning in Java programming.
2. Discuss the importance of summaries, glossary, and exercises in reinforcing concepts.

UNIT – 17
INHERITANCE

UNIT STRUCTURE

17.1 Introduction
17.2 Objective
17.3 Types of Inheritance
17.4 Inheritance Mechanism
17.5 Summary
17.6 Glossary
17.7 Answer to check your progress
17.8 Suggested readings - OER
17.9 Reference
17.10 Exercise

17.1. INTRODUCTION

In the earlier units, learners studied important object-oriented programming concepts in Java such as classes, objects, methods, constructors, overloading, static members, and memory management. These topics form the backbone of Java programming and enable students to design structured, modular, and efficient programs. As learners move further in the syllabus, it becomes essential to consolidate this foundational knowledge and prepare for more advanced programming constructs that emphasize reliability, scalability, and real-world application development.

Unit 17 is designed to strengthen conceptual continuity and reinforce disciplined programming practices. This unit encourages learners to reflect on previously learned concepts and understand how they integrate into larger Java applications. Programming is not only about writing correct code but also about understanding execution flow, proper resource usage, and applying best practices to solve problems effectively. The self-learning material (SLM) structure of this unit provides clear objectives, concise summaries, and academic support tools such as glossary, exercises, and suggested readings. These components help learners revise key ideas, assess their understanding, and explore additional learning resources independently. By engaging actively with this unit, students can improve their analytical ability and confidence in applying Java programming concepts. By the end of Unit 17, learners are expected to be better prepared to understand advanced Java topics, analyze

NOTES

program behavior, and approach complex programming tasks with a structured and logical mindset. Thus, Unit 17 serves as an important academic link in the progression of Java programming studies.

17.2. OBJECTIVES

After completing this unit, the learner will be able to:

- Recall core Java programming concepts studied in previous units
- Identify important terminology and ideas related to this unit
- Explain the relevance of structured learning in Java programming
- Apply previously learned concepts in basic programming scenarios
- Analyze program logic and execution behavior
- Develop systematic and disciplined problem-solving skills

17.3. TYPES OF INHERITANCE

Inheritance is one of the most important pillars of Object-Oriented Programming (OOP) in Java. It allows a class to acquire the properties and behavior of another class, thereby promoting code reusability, extensibility, and maintainability. Inheritance models real-world relationships, such as "is-a" relationships, where one entity is a specialized form of another.

In Java, inheritance is implemented using the extends keyword. The class whose properties are inherited is called the superclass (parent class), and the class that inherits those properties is called the subclass (child class). Through inheritance, common functionality can be written once in a superclass and reused in multiple subclasses, reducing redundancy and improving consistency.

Java supports several types of inheritance at the class level, though with certain restrictions to maintain simplicity and avoid ambiguity. Understanding the different types of inheritance helps students design efficient class hierarchies and write scalable programs.

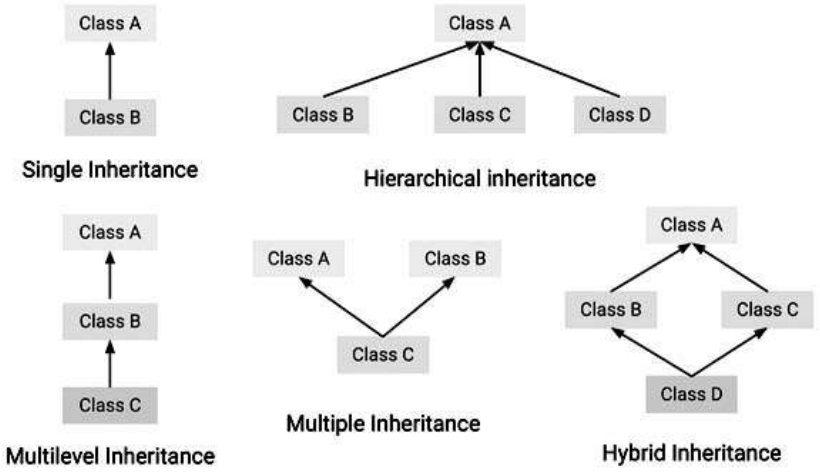


Fig. 17.1 : Types of Inheritance.

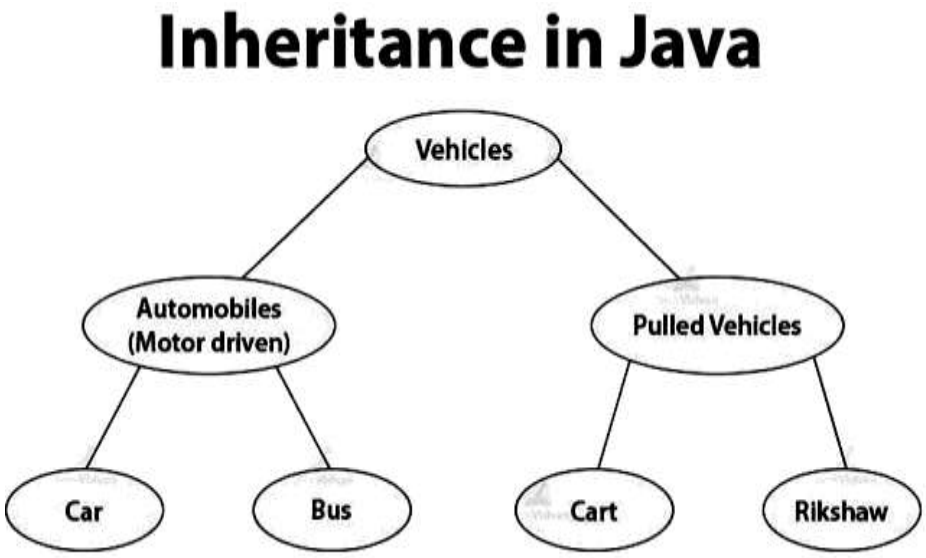


Fig. 17.2 : Inheritance in Java.

17.3.1. Single Inheritance

Single inheritance is the simplest and most used form of inheritance in Java. In this type, a single subclass inherits from a single superclass. This establishes a direct parent–child relationship between two classes.

Concept Explanation

In single inheritance:

- One parent class
- One child class
- Clear and simple class structure

Syntax

```
class Parent {
    void show() {
        System.out.println("Parent class method");
    }
}

class Child extends Parent {
    void display() {
        System.out.println("Child class method");
    }
}
```

Usage

```
Child c = new Child();
c.show();
c.display();
```

Advantages of Single Inheritance

- Simple and easy to understand
- Improves code reusability

NOTES

- Enhances readability
- Reduces program complexity

Limitations

- Limited flexibility compared to multiple inheritance
- Subclass depends heavily on one superclass

Single inheritance is widely used in Java applications such as student–person, employee–person, and savings Account–account relationships.

17.3.2. Multilevel Inheritance

Multilevel inheritance occurs when a class is derived from another class, which itself is derived from another class. This forms a chain of inheritance, where properties and methods flow from the topmost class down to the lowest subclass.

Concept Explanation

In multilevel inheritance:

- One base class
- One or more intermediate classes
- One final derived class

Syntax

```
class A {
    void showA() {
        System.out.println("Class A");
    }
}

class B extends A {
    void showB() {
        System.out.println("Class B");
    }
}

class C extends B {
    void showC() {
        System.out.println("Class C");
    }
}
```

Usage

```
C obj = new C();
obj.showA();
obj.showB();
obj.showC();
```

Advantages

- Reuse of code at multiple levels
- Logical class hierarchy
- Supports incremental development

Limitations

- Increased dependency between classes
- Debugging becomes complex
- Tight coupling if not designed carefully

Multilevel inheritance is commonly used in layered application designs and role-based systems.

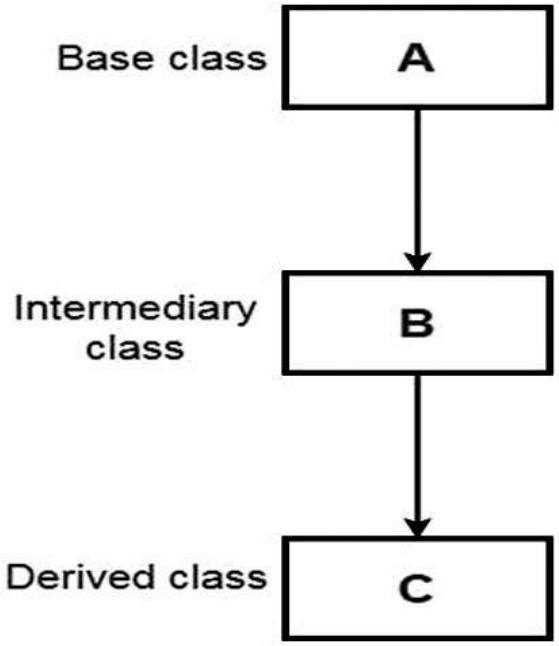


Fig. 17.3 : Multilevel Inheritance.

17.3.3. Hierarchical Inheritance

Hierarchical inheritance occurs when multiple subclasses inherit from a single superclass. This allows common functionality to be shared among several child classes.

Concept Explanation

In hierarchical inheritance:

- One parent class
- Multiple child classes
- Each child can have unique behavior

Syntax

```
class Vehicle {
    void fuel() {
        System.out.println("Uses fuel");
    }
```

NOTES

```
}

class Car extends Vehicle {
    void wheels() {
        System.out.println("Four wheels");
    }
}

class Bike extends Vehicle {
    void wheels() {
        System.out.println("Two wheels");
    }
}
```

Advantages

- High code reusability
- Centralized common logic
- Easy maintenance

Limitations

- Changes in superclass affect all subclasses
- Overuse can create rigid structures

Hierarchical inheritance is widely used in frameworks, UI components, and domain modeling.

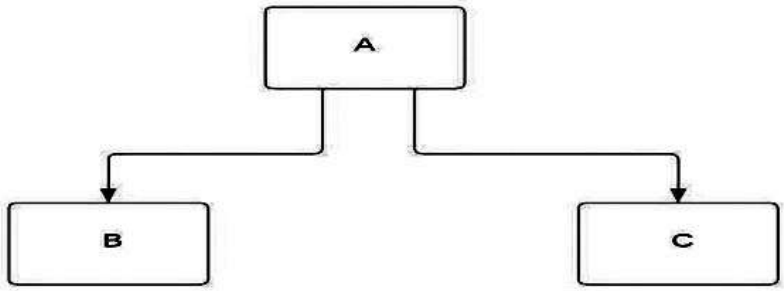


Fig. 17.4 : Hierarchical Inheritance.

17.4. INHERITANCE MECHANISM

The inheritance mechanism in Java defines how data members and methods of a superclass are made available to a subclass. Java follows a controlled inheritance model, ensuring security and clarity in class relationships.

Key points of Java inheritance mechanism:

- Only public and protected members are inherited
- Private members are not directly accessible
- Method overriding enables runtime polymorphism
- Constructors are not inherited

Understanding this mechanism is essential to avoid misuse of inherited members and to design robust class hierarchies.

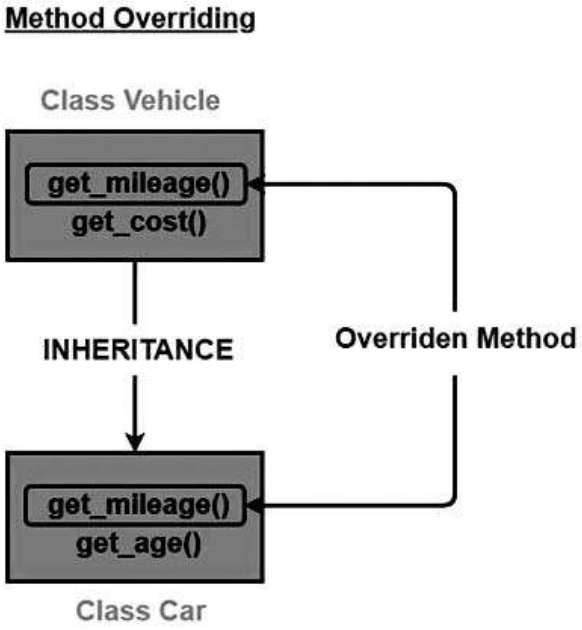


Fig. 17.5 : Inheritance Mechanism.

Access Modifier	Within Class	Within Package	Outside package without inheritance	Outside package with inheritance
Default	✓	✓	✘	✘
Private	✓	✘	✘	✘
Protected	✓	✓	✘	✓
Public	✓	✓	✓	✓

17.4.1. super Keyword

The super keyword in Java is used to refer to the immediate parent class object. It helps in accessing superclass members that are hidden or overridden in the subclass.

Uses of super Keyword

1. Accessing Superclass Variables

```
class Parent {
    int x = 10;
}
class Child extends Parent {
    int x = 20;

    void show() {
        System.out.println(super.x);
    }
}
```

NOTES

2. Calling Superclass Methods

```
super.show();
```

3. Calling Superclass Constructor

```
super();
```

Advantages

- Resolves naming conflicts
- Ensures proper constructor execution
- Improves clarity in inheritance

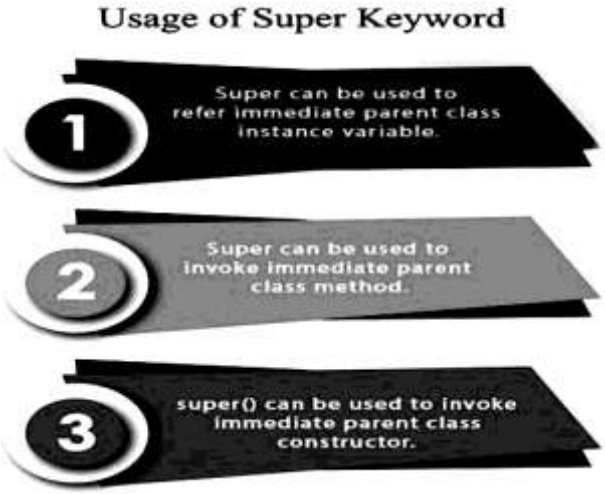


Fig. 17.6 : Super Keyword.

17.4.2. Constructor Execution Order

In inheritance, constructors are executed in a specific order. When an object of a subclass is created, the superclass constructor is executed first, followed by the subclass constructor.

Activity 17.1. Organizational Hierarchy Design

Objectives:

- Understand inheritance through real-world hierarchy
- Identify common and specialized roles

Materials Needed:

- Notebook or digital tool

Activity Description:
Create a class hierarchy with *Employee* as the base class and at least two derived classes (e.g., Manager, Intern).
List common attributes and specific features for each. Draw a simple diagram.

Reflection/Outcome:
Write 2–3 lines on how inheritance reduces duplication.

Execution Order Rules

1. Superclass constructor is invoked
2. Subclass constructor is executed
3. This ensures base initialization first

Example

```
class A {
    A() {
        System.out.println("Class A Constructor");
    }
}

class B extends A {
    B() {
        System.out.println("Class B Constructor");
    }
}
```

Output

Class A Constructor

Class B Constructor

☑ Check Your Progress 17.1.

1. Explain the different types of inheritance in Java.

   . . . . . .

   . . . . . .

   . . . . . .

2. What is the role of the super keyword in inheritance?

   . . . . . .

   . . . . . .

   . . . . . .

3. Explain the order of constructor execution in inheritance.

   . . . . . .

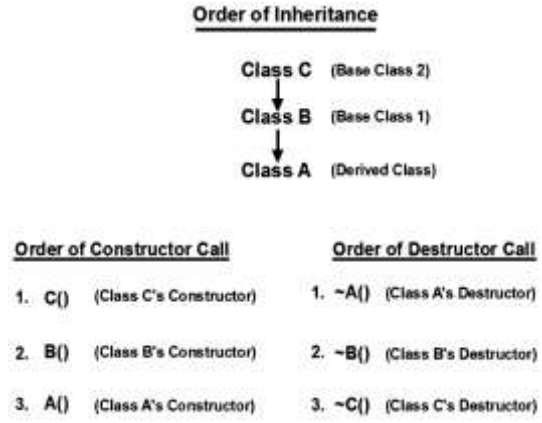
   . . . . . .

   . . . . . .

Importance

- Ensures proper initialization
- Prevents inconsistent object state
- Supports inheritance reliability

NOTES



17.5. SUMMARY

This unit reinforces the connection between previously learned Java concepts and upcoming advanced topics. Clearly defined objectives guide learners toward measurable learning outcomes. Emphasis is placed on logical reasoning and structured program design. The unit supports revision of foundational concepts to strengthen understanding. Academic tools such as glossary and exercises enhance self-learning. Self-assessment enables learners to evaluate comprehension and progress. Suggested readings encourage independent and extended learning. Overall, the unit strengthens the academic base required for advanced Java programming.

17.6. GLOSSARY

- **Objective –** A clearly defined and measurable learning outcome
- **Concept Reinforcement –** Strengthening previously learned programming ideas
- **Syntax –** Rules that define the structure of Java statements
- **Logic –** Step-by-step reasoning used to solve programming problems
- **Compilation –** Conversion of Java source code into bytecode
- **Execution Flow –** Order in which program instructions are executed
- **Runtime –** Phase during which a program is running
- **Error –** A mistake that leads to incorrect program behavior
- **Debugging –** Process of identifying and correcting errors

17.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** Inheritance in Java allows one class to acquire the properties and methods of another class. The main types are:
   - Single inheritance: One class inherits from one parent class.
   - Multilevel inheritance: A class is derived from another derived class, forming a chain.

- Hierarchical inheritance: Multiple classes inherit from a single parent class.

2. Answer : The super keyword is used to refer to the immediate parent class. It can be used to access parent class variables, call parent class methods, and invoke the parent class constructor.

3. Answer : In inheritance, constructors are executed in a specific order. First, the parent class constructor is called, followed by the child class constructor. This ensures that the parent class is properly initialized before the child class uses its properties.

17.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3758	

17.9. REFERENCES

- Balagurusamy, E. (2022). *Programming with Java*. McGraw-Hill Education.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Oracle. (2023). Java Documentation.
- NPTEL. (2022). Java Programming Courses.

17.10 EXERCISE

Part 1

A. Multiple Choice Question

1. Inheritance in Java is used to:

   (a) Hide data (c) Create loops

   (b) Reuse code from another class (d) Define variables

2. Single inheritance means:

   (a) One class inherits from multiple classes

   (b) One class inherits from one parent class

   (c) Multiple classes inherit from one class

   (d) No inheritance

3. Multilevel inheritance involves:

   (a) One class only (b) A chain of inheritance

   (c) Independent classes (d) No parent class

4. The super keyword is used to:

NOTES

(a) Create objects (b) Access parent class members

(c) Define methods (d) Delete variables

5. In inheritance, constructors are executed in:

(a) Child to parent order (b) Random order

(c) Parent to child order (d) No specific order

Answers with Explanations

1. Ans. (b)

Inheritance Concept : Inheritance allows a class to reuse properties and methods of another class, promoting code reusability.

2. Ans. (b)

Single Inheritance : In single inheritance, one class inherits from one parent class.

3. Ans. (b)

Multilevel Inheritance : Multilevel inheritance forms a chain where a class inherits from another derived class.

4. Ans. (b)

super Keyword : The super keyword is used to access parent class variables, methods, and constructors.

5. Ans. (c)

Constructor Execution : Constructors are executed from parent to child, ensuring proper initialization of inherited properties.

B. Short Answer Questions (50–100 words)

1. What is inheritance in Java?
2. Explain single inheritance with an example.
3. What is multilevel inheritance?
4. What is the use of the super keyword?
5. Explain constructor execution order in inheritance.

Part 2

C. Long Answer Questions (200 words)

1. Explain different types of inheritance in Java.
2. Discuss the inheritance mechanism and the role of the super keyword.

Part 3

D. Long Answer Questions (300 words)

1. Explain single, multilevel, and hierarchical inheritance with examples.

2. Discuss constructor execution order and use of the super keyword in detail.

NOTES

UNIT – 18
POLYMORPHISM AND ABSTRACTION

UNIT STRUCTURE

18.1 Introduction
18.2 Objective
18.4 Abstract Classes and Containership
18.5 Summary
18.6 Glossary
18.7 Answer to check your progress
18.8 Suggested readings - OER
18.9 Reference
18.10 Exercise

18.1. INTRODUCTION

In the previous units, learners have studied advanced object-oriented programming concepts in Java, including inheritance and its various forms. These concepts help in designing reusable, extensible, and well-structured programs. As Java applications grow and complexity, it becomes increasingly important to understand how different programming constructs work together to ensure clarity, maintainability, and correctness. Unit 18 is designed to reinforce conceptual understanding and prepare learners for further advanced Java topics. This unit emphasizes disciplined programming practices, logical organization of concepts, and systematic learning progression. By revisiting and strengthening previously acquired knowledge, learners can develop a deeper understanding of how Java programs are structured and executed.

The self-learning material (SLM) approach followed in this unit provides clearly defined objectives, concise summaries, and academic support tools such as glossary, exercises, and suggested readings. These components enable learners to evaluate their understanding, revise essential ideas, and independently explore additional learning resources. By the end of Unit 18, learners are expected to demonstrate improved confidence in applying Java programming concepts, analyzing program behavior, and approaching problem-solving tasks with a structured and logical mindset. Thus, Unit 18 serves as an important academic link in the continuous learning process of Java programming.

NOTES

18.2. OBJECTIVES

After completing this unit, the learner will be able to:

- **Recall** relevant Java programming concepts studied in earlier units
- **Identify** key terminology and ideas related to this unit
- **Explain** the importance of structured learning in Java programming
- **Apply** previously learned concepts in simple programming situations
- **Analyze** program logic and execution flow
- **Develop** systematic and disciplined problem-solving skills

18.3. METHOD OVERRIDING AND FINAL KEYWORD

In object-oriented programming, inheritance allows a subclass to acquire the properties and methods of a superclass. However, in many real-world situations, a subclass may need to modify or redefine the behavior of an inherited method to suit its specific requirements. Java supports this through method overriding. Alongside overriding, Java provides the final keyword, which restricts modification and inheritance to ensure security, consistency, and controlled program behavior.

Method overriding and the final keyword play a crucial role in achieving runtime polymorphism, one of the most powerful features of Java. These concepts allow programs to be flexible while also providing mechanisms to prevent unwanted changes.

18.3.1. Method Overriding

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The overridden method in the subclass must have the same method name, parameter list, and return type as the method in the superclass.

Concept Explanation

In method overriding:

- The method call is resolved at runtime
- The object type (not reference type) determines which method is executed
- This behavior is known as dynamic binding

Syntax Example

```
class Parent {
    void show() {
        System.out.println("Parent class
method");
```

```
    }
}

class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child class method");
    }
}
```

Usage

```
Parent p = new Child();
p.show();
```

Output:

```
Child class method
```

This demonstrates runtime polymorphism, where the method call depends on the object type.

- Rules for Method Overriding
- Method name must be the same
- Parameter list must be the same
- Access level cannot be more restrictive
- Static methods cannot be overridden
- Constructors cannot be overridden

Advantages of Method Overriding

- Supports runtime polymorphism
- Improves flexibility of code
- Allows specialization of behavior
- Enhances extensibility

Limitations

- Improper overriding can cause logical errors
- Tight coupling between superclass and subclass

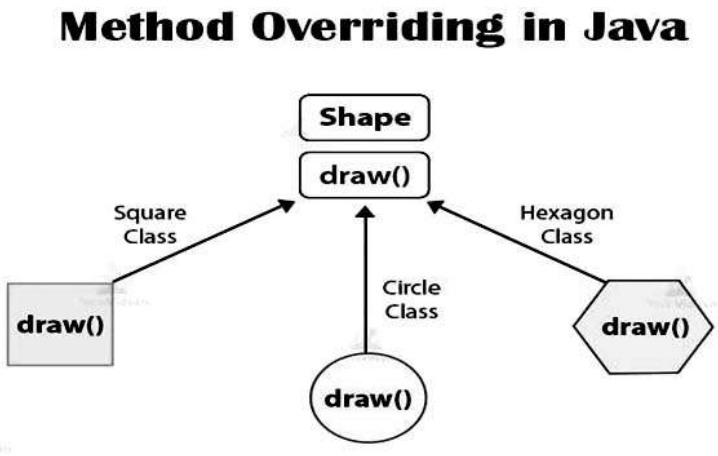


Fig. 18.1 : Method Overriding

NOTES

18.3.2. Final Variables and Methods

The final keyword in Java is used to restrict modification. It can be applied to variables, methods, and classes. In this section, we focus on final variables and final methods.

Final Variables

A final variable is a variable whose value cannot be changed once assigned.

```
final int MAX = 100;
```

Key points:

- Final variables act as constants
- Must be initialized only once
- Improve program safety and readability

Final Methods

A final method is a method that cannot be overridden by subclasses.

```
class Base {
    final void display() {
System.out.println("This method cannot be overridden");
    }
}
```

Why Use Final Methods

- Prevent alteration of critical logic
- Ensure consistency across subclasses
- Improve security

Advantages of Final Keyword

- Prevents accidental modification
- Enhances program reliability
- Improves performance (in some cases)

Limitations

- Reduces flexibility
- Overuse can limit extensibility

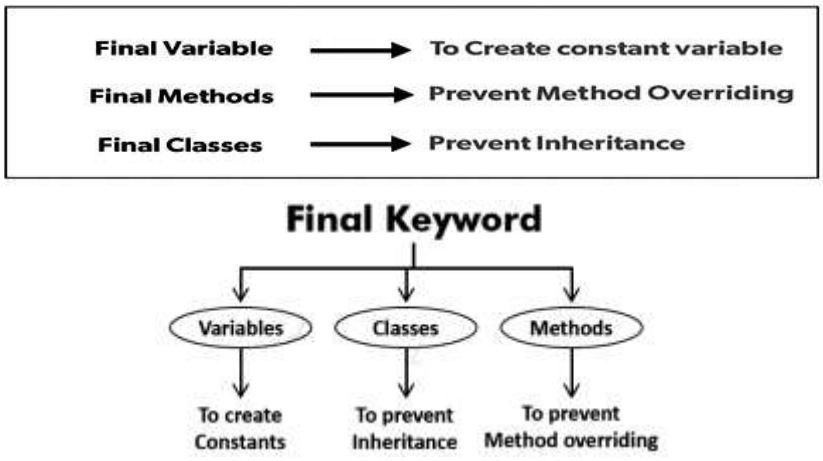


Fig. 18.2 : Final Keyword.

18.4. ABSTRACT CLASSES AND CONTAINERSHIP

Java provides advanced mechanisms to achieve abstraction and code organization. Two such mechanisms are abstract classes and containership. These concepts help programmers design programs that are modular, reusable, and easy to maintain. Abstract classes define what an object should do, while containership defines how objects are related and organized with in a program.

18.4.1. Abstract Classes

An abstract class is a class that cannot be instantiated and may contain abstract methods (methods without a body) as well as concrete methods.

Syntax

```
abstract class Shape
{
    abstract void draw();
}
```

Subclass Implementation

```
class Circle extends Shape
{
    void draw() {
        System.out.println("Drawing Circle");
    }
}
```

Key Characteristics

- Declared using abstract keyword
- Can contain both abstract and non-abstract methods
- Supports inheritance
- Used to achieve abstraction

Advantages

- Forces subclasses to implement required methods
- Provides partial abstraction
- Improves design consistency

Limitations

- Cannot create objects directly
- Multiple inheritance not supported through classes

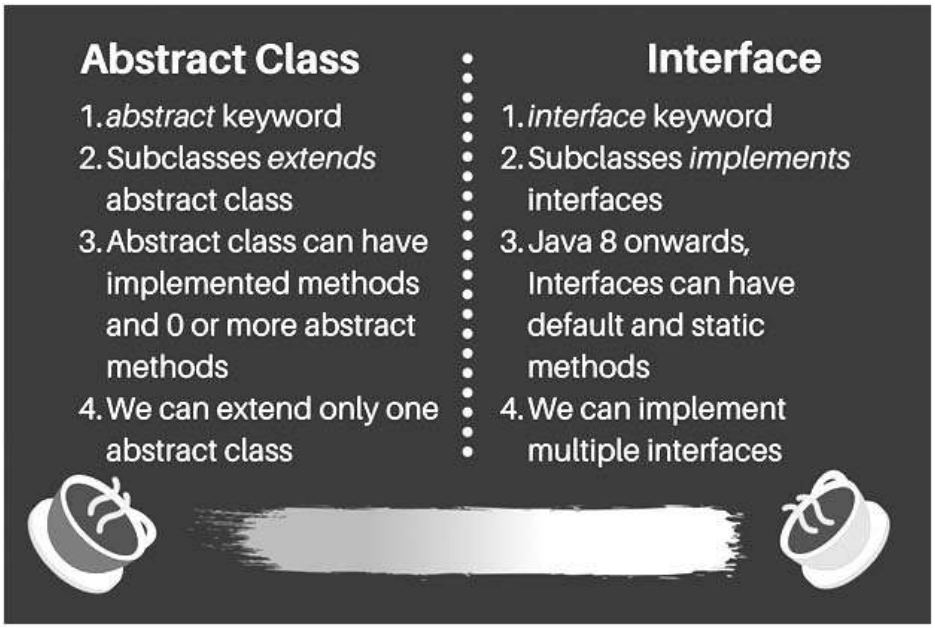


Fig. 18.3 : Abstract Classes

18.4.2. Containership

Containership (also known as composition or aggregation) represents a relationship where one class contains objects of another class. This **is known as a "has-a" relationship.**

Example

```
class Address {
    String city;
}

class Student {
    Address addr;
}
```

Here, a Student **has an** Address.

Types of Containership

- **Aggregation –** Weak association
- **Composition –** Strong association

Advantages

- Promotes code reuse
- Improves modularity
- Represents real-world relationships

Comparison with Inheritance

Inheritance	Containership
Is-a relationship	Has-a relationship
Tight coupling	Loose coupling
Less flexible	More flexible

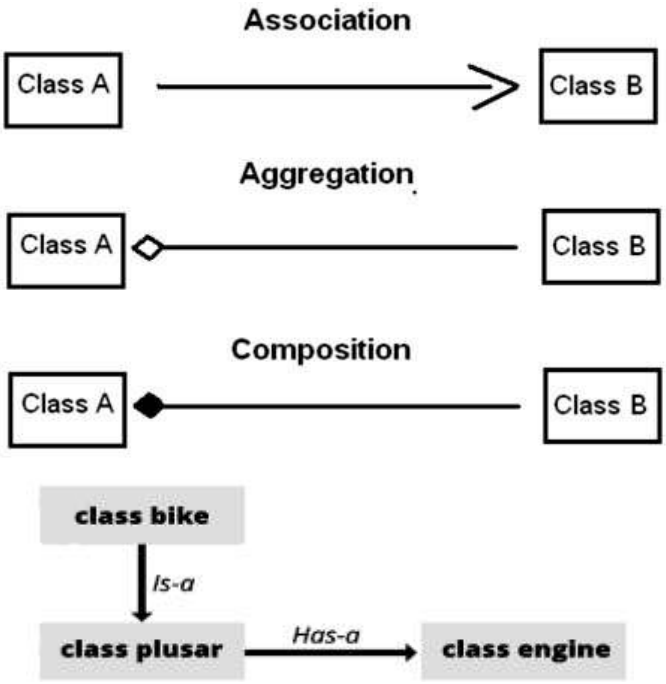


Fig. 18.4 : Containership.

18.4.3. Visibility Control

Visibility control in Java determines which class members are accessible in different parts of a program. It is achieved using access specifiers.

Access Specifiers

- **public –** Accessible everywhere
- **protected –** Accessible within package and subclasses
- **default –** Accessible within the same package
- **private –** Accessible only within the class

Importance of Visibility Control

- Protects data
- Supports encapsulation
- Prevents misuse of class members
- Improves maintainability

Example

```
class Account {
    private double balance;

    public double getBalance() {
        return balance;
    }
}
```

This ensures controlled access to sensitive data.

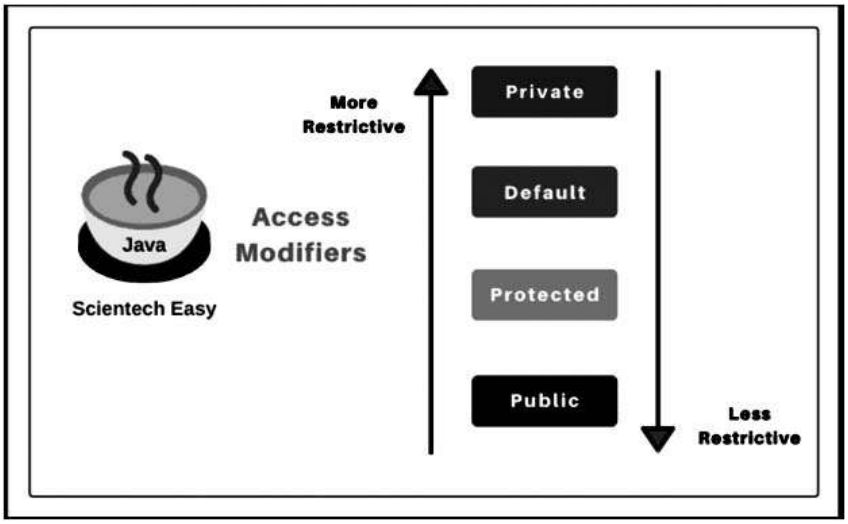


Fig. 18.5: Visibility Control.

Access Modifiers

Modifier	Class	Package	SubClass	Global
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
private	✓	✗	✗	✗

Fig. 18.6: Access Modifiers.

Activity 18.1. Designing a Business Model Using Abstract Classes

Objectives:
To understand how abstract classes provide a blueprint for real-world business models and enforce consistency in implementation.

Materials Needed:
Notebook or digital document, basic knowledge of any programming language (preferably Java/Python).

Activity Description:
Imagine you are designing a system for different types of employees in a company (e.g., Manager, Intern, Consultant).

1. Create an abstract class named Employee with abstract methods such as calculate Salary() and getRoleDescription().

2. Define at least two subclasses that extend the abstract class and implement these methods differently.
3. Write pseudocode or actual code showing how each subclass behaves uniquely.
4. Analyze how the abstract class ensures a common structure across all employee types.

Reflection/Outcome:
Briefly write how abstract classes help in maintaining uniformity while allowing flexibility in business applications. Identify one real-world scenario in management where this concept can be applied.

☑ **Check Your Progress 18.1.**

1. What is method overriding? How does it differ from method overloading?

   . . . . . .
   . . . . . .
   . . . . . .

2. What is the purpose of the `final` keyword in variables and methods?

   . . . . . .
   . . . . . .
   . . . . . .

3. What are abstract classes and containership? Explain visibility control in Java.

   . . . . . .
   . . . . . .
   . . . . . .

18.5. SUMMARY

This unit reinforces continuity between earlier Java concepts and upcoming advanced topics. Clearly stated objectives guide learners toward measurable learning outcomes. Emphasis is placed on logical reasoning and structured program design.

The unit supports revision of foundational concepts to strengthen understanding. Academic tools such as glossary and exercises enhance self-learning. Self-assessment helps learners evaluate comprehension and progress. Suggested readings promote independent and extended learning. Overall, the unit strengthens the academic base required for advanced Java programming.

18.6. GLOSSARY

- **Objective –** A clearly defined and measurable learning outcome

NOTES

- **Concept Reinforcement –** Strengthening previously learned programming ideas
- **Syntax –** Rules that define the structure of Java statements
- **Logic –** Step-by-step reasoning used to solve programming problems
- **Compilation –** Process of converting Java source code into bytecode
- **Execution Flow –** Order in which program instructions are executed
- **Runtime –** Phase during which a program is running
- **Error –** A mistake that leads to incorrect program behavior
- **Debugging –** Process of identifying and correcting errors
- **Self-Learning Material (SLM) –** Structured content designed for independent study

18.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its parent class with the same name and parameters. It is used to achieve runtime polymorphism. Unlike method overloading, which occurs within the same class and differs in parameters, overriding involves inheritance and the same method signature.
2. **Answer :** The final keyword is used to restrict modification. A final variable cannot be changed once assigned, making it constant. A final method cannot be overridden by subclasses, ensuring that its behavior remains unchanged.
3. **Answer :** An abstract class is a class that cannot be instantiated and may contain abstract methods (methods without a body). It is used as a base class for other classes.Containership refers to a “has-a” relationship where one class contains objects of another class, enabling code reuse.Visibility control refers to access modifiers (public, private, protected, default) that determine the accessibility of class members, ensuring proper encapsulation and security.

18.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3759	

18.9. REFERENCES

- Balagurusamy, E. (2022). *Programming with Java*. McGraw-Hill Education.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Oracle. (2023). *Java Documentation.*
- NPTEL. (2022). *Java Programming Courses*.

NOTES

18.10. EXERCISE

Part 1

A. Multiple Choice Question

1. Method overriding occurs when:
   (a) Methods have different names
   (b) A subclass provides its own implementation of a parent class method
   (c) Methods are deleted
   (d) Methods are static
2. The final keyword is used to:
   (a) Modify variables anytime
   (b) Restrict modification
   (c) Create loops
   (d) Define packages
3. A final variable is:
   (a) Changeable
   (b) Constant
   (c) Temporary
   (d) Undefined
4. An abstract class is:
   (a) A class that can be instantiated
   (b) A class that cannot be instantiated
   (c) A method
   (d) A variable
5. Containership represents:
   (a) Inheritance relationship
   (b) “has-a” relationship
   (c) Loop structure
   (d) Conditional statement

Answers with Explanations

1. Ans. (b)

NOTES

Method Overriding : Method overriding occurs when a subclass provides a specific implementation of a method already defined in the parent class.

2. Ans. (a)

Final Keyword : The final keyword is used to restrict modification of variables, methods, or classes.

3. Ans. (b)

Final Variable : A final variable acts as a constant and cannot be changed once assigned.

4. Ans. (b)

Abstract Class : An abstract class cannot be instantiated and is used as a base for other classes.

5. Ans. (b)

Containership : Containership represents a "has-a" relationship where one class contains an object of another class.

B. Short Answer Questions (50–100 words)

1. What is method overriding?
2. What is the purpose of the final keyword?
3. What are abstract classes?
4. Explain containership in Java.
5. What is visibility control?

Part 2

C. Long Answer Questions (200 words)

1. Explain method overriding and the use of the final keyword in Java.
2. Discuss abstract classes and containership in Java.

Part 3

D. Long Answer Questions (300 words)

1. Explain method overriding with examples and discuss final variables and methods.
2. Discuss abstract classes, containership, and visibility control in detail.

UNIT – 19
INTERFACES, PACKAGES AND EXCEPTIONS

UNIT STRUCTURE

19.1 Introduction
19.2 Objective
19.3 Interfaces and Packages
19.4 Exception Handling
19.5 Summary
19.6 Glossary
19.7 Answer to check your progress
19.8 Suggested readings - OER
19.9 Reference
19.10 Exercise

19.1. INTRODUCTION

In earlier units, learners explored advanced object-oriented programming concepts in Java, including inheritance, abstract classes, method overriding, and visibility control. These concepts help programmers design reusable, extensible, and well-structured software systems. As Java applications grow and complexity, it becomes increasingly important to organize code efficiently and handle unexpected situations that may arise during program execution.

Unit 19 is designed to reinforce conceptual continuity and prepare learners for practical challenges encountered in real-world Java programming. This unit emphasizes structured learning, disciplined coding practices, and the importance of robust program design. By building upon previously learned concepts, students are encouraged to understand how different programming mechanisms work together to improve reliability, maintainability, and scalability. The self-learning material (SLM) approach followed in this unit provides clearly defined objectives, concise summaries, and academic support tools such as glossary, exercises, and suggested readings. These components help learners revise key ideas, evaluate their understanding, and independently explore additional learning resources. Through this unit, students are expected to strengthen their analytical ability and confidence in applying Java programming concepts effectively. By the end of Unit 19, learners will be better prepared to approach advanced

NOTES

Java topics with a systematic and logical mindset, making this unit an important step in the overall learning progression of Java programming.

19.2. OBJECTIVES

After completing this unit, the learner will be able to:

- **Recall** relevant Java programming concepts studied in previous units
- **Identify** important ideas and terminology related to this unit
- **Explain** the significance of structured learning in Java programming
- **Apply** previously learned concepts in basic programming scenarios
- **Analyze** program logic and execution behavior
- **Develop** systematic and disciplined problem-solving skills

19.3. INTERFACES AND PACKAGES

As Java programs grow larger and more complex, developers need mechanisms that support abstraction, multiple inheritance–like behavior, modularity, and code organization. Java provides two powerful features for this purpose: interfaces and packages. Interfaces allow programmers to define what a class must do, without specifying how it should be done, while packages help in grouping related classes and interfaces into a logical unit. Interfaces promote loose coupling and standardization, making programs more flexible and extensible. Packages, on the other hand, improve code readability, reusability, and security by organizing large projects into manageable components. Together, interfaces and packages play a crucial role in professional Java application development.

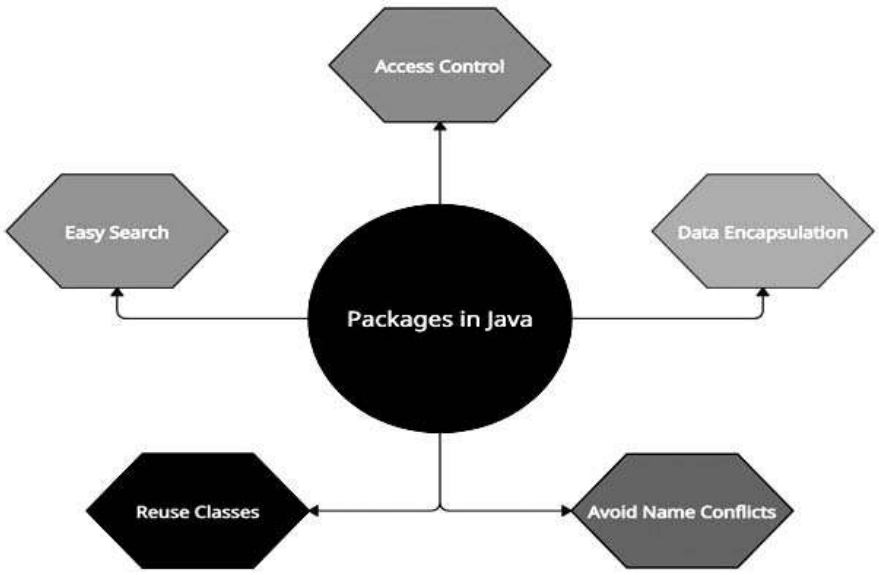


Fig. 19.1 : Interfaces And Packages.

19.3.1. Defining Interfaces

In Java, an interface is a powerful mechanism used to achieve abstraction, standardization, and loose coupling in object-oriented programming. An interface defines what a class should do, but not how it should do it. It acts as a contract that specifies a set of methods that a class must implement. Interfaces play a crucial role in designing large, flexible, and maintainable Java applications.

Unlike classes, interfaces do not contain method implementations (with limited modern exceptions). They only declare method signatures and constants. This separation of specification from implementation allows different classes to implement the same interface in their own way. As a result, interfaces support polymorphism, enhance code reusability, and enable Java to overcome the limitation of multiple inheritance in classes. In real-world programming, interfaces are used to define common behavior across unrelated classes. For example, interfaces such as Runnable, Comparable, and Serializable are widely used in Java libraries to ensure consistency across different implementations. Understanding how to define interfaces is therefore essential for professional Java programming.

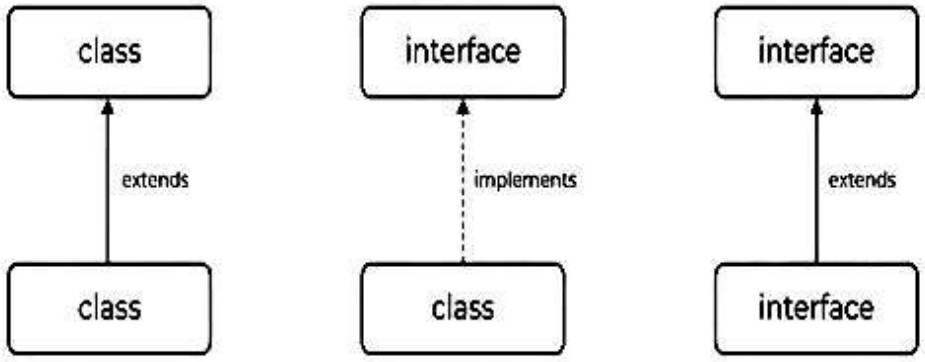


Fig. 19.2 : Defining Interfaces.

Concept of Interface

An interface can be described as a blueprint of a class that contains:

- Abstract method declarations
- Public static final variables (constants)

An interface cannot be instantiated, meaning objects of an interface cannot be created directly. Instead, a class must implement the interface and provide implementations for its methods.

Interfaces help in:

- Achieving complete abstraction
- Defining standard methods
- Supporting multiple inheritance
- Designing loosely coupled systems

Syntax for Defining an Interface

An interface is defined using the interface keyword.

NOTES

```
interfaceInterfaceName {
// method declarations
}
Example
interfaceShape {
voiddraw();
}
```

In this example:

- Shape is an interface
- draw() is an abstract method
- No method body is provided

By default:

- All methods are public and abstract
- All variables are public, static, and final

Declaring Methods in an Interface

Methods in an interface are declared without implementation.

```
interfaceCalculator {
intadd(int a, int b);
intsubtract(int a, int b);
}
```

These methods define a contract that any implementing class must follow. A class implementing this interface must provide method bodies for both add() and subtract().

Declaring Variables in an Interface

Variables declared inside an interface are treated as constants.

```
interfaceLimits {
intMAX=100;
}
```

This is equivalent to:

```
publicstaticfinalintMAX=100;
```

Key points:

- Values cannot be changed
- Useful for defining fixed configuration values

Characteristics of Interfaces

Interfaces in Java have the following important characteristics:

- Declared using the interface keyword
- Cannot be instantiated
- Do not have constructors
- Support multiple inheritance

- Can extend other interfaces
- Used to achieve abstraction

These characteristics distinguish interfaces from classes and make them suitable for defining contracts.

Extending Interfaces

An interface can extend one or more interfaces using the extends keyword.

```
interfaceA {
voidmethodA();
}

interfaceBextendsA {
voidmethodB();
}
```

Here, interface B inherits all methods of interface A. This supports multiple inheritance at the interface level, which is not allowed in classes.

Why Defining Interfaces Is Important

Defining interfaces provides several advantages:

- Encourages standardization
- Enables multiple inheritance
- Improves code flexibility
- Supports runtime polymorphism
- Reduces dependency between classes

Interfaces are extensively used in:

- Framework design
- API development
- Enterprise applications
- Event-driven programming

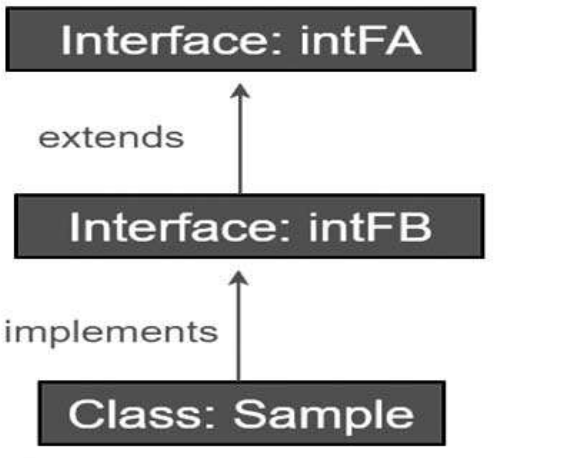


Fig. 19.3 :

NOTES

Difference Between Interface and Class

Interface	Class
Supports multiple inheritance	Does not support multiple inheritance
Only method declarations	Method implementation allowed
No constructors	Constructors allowed
Used for abstraction	Used for implementation

19.3.2. Implementing Interfaces

A class implements an interface using the implements keyword. When a class implements an interface, it must provide implementations for all abstract methods defined in the interface.

Syntax

```
class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}
```

Using the Interface

```
Shape s = new Circle();
s.draw();
```

This demonstrates runtime polymorphism, where the interface reference refers to an object of the implementing class.

Multiple Interfaces

Java allows a class to implement multiple interfaces, solving the limitation of multiple inheritance in classes.

```
class Demo implements A, B {
    // method implementations
}
```

Advantages of Implementing Interfaces

- Promotes loose coupling
- Improves flexibility
- Supports polymorphism
- Enhances code reusability

19.3.3. User-Defined Packages

A package in Java is a namespace that groups related classes and interfaces together. Packages help in organizing large programs and avoiding name conflicts.

Types of Packages

- Built-in packages (java.lang, java.util, etc.)
- User-defined packages

Creating a User-Defined Package

```
package mypackage;

public class Test {
    public void show() {
      System.out.println("User-defined package example");
    }
}
```

Accessing a Package

```
import mypackage.Test;
```

Advantages of Packages

- Code organization
- Reusability
- Security through access control
- Avoids naming conflicts

Packages are essential in enterprise-level applications where code modularity is critical.

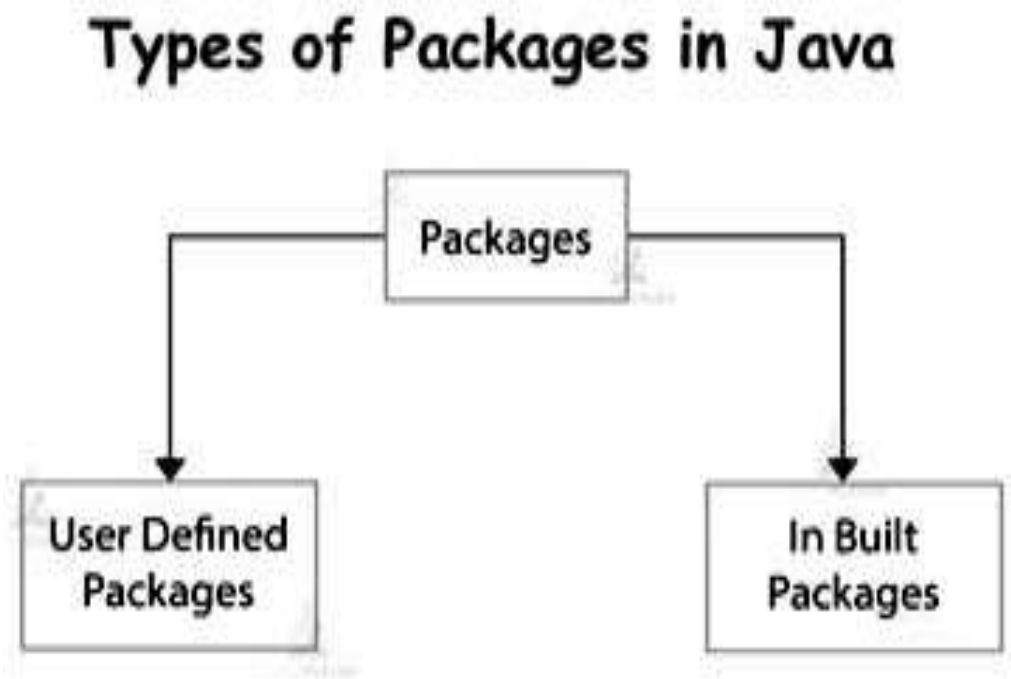


Fig. 19.4 : Types of packages in Java.

19.4. EXCEPTION HANDLING

In real-world applications, errors may occur during program execution due to invalid input, resource unavailability, or logical mistakes. Java provides exception handling to manage such runtime errors gracefully

NOTES

without crashing the program. Exception handling separates normal program logic from error-handling logic, improving program reliability and maintainability.

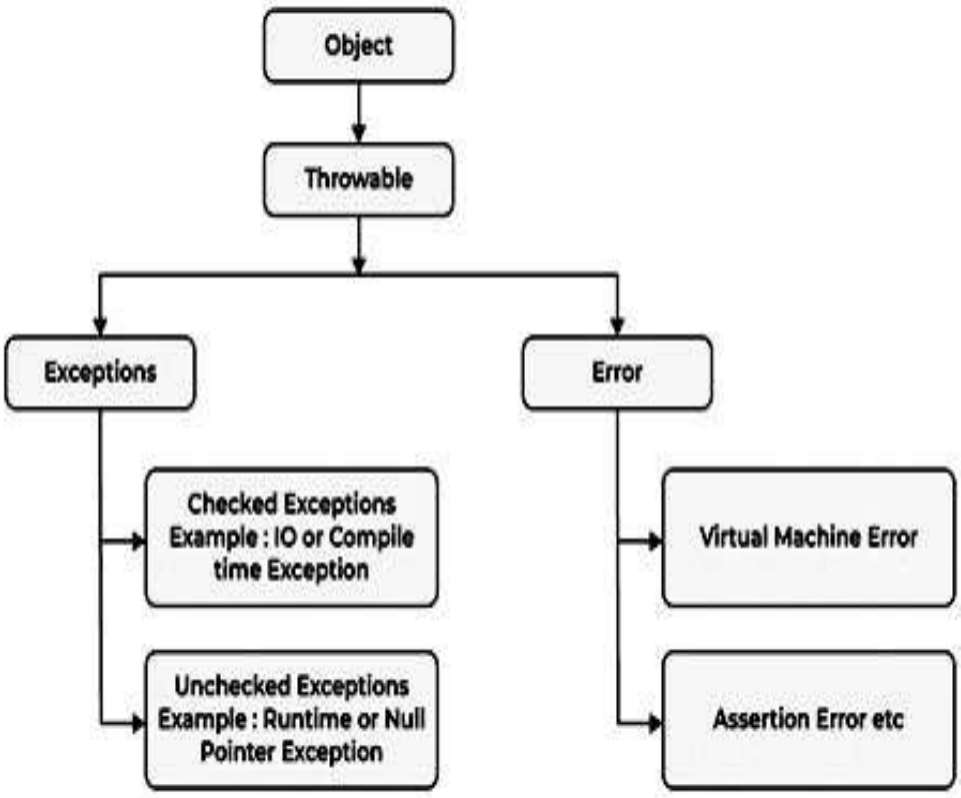


Fig. 19.5 : Exception Handling.

19.4.1. try–catch Block

The try–catch block is the basic mechanism used to handle exceptions in Java.

Syntax

```
try {
    // risky code
} catch (Exception e) {
    // handling code
}
```

Example

```
try {
    int a = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Division by zero error");
}
```

Advantages

- Prevents abnormal termination

- Improves program robustness
- Provides meaningful error messages

19.4.2. Multiple Catch

Java allows multiple catch blocks to handle different types of exceptions separately.

Syntax

```
try {
    // code
} catch (ArithmeticException e) {
    // handle arithmetic error
} catch (NullPointerException e) {
    // handle null error
}
```

Rules

- More specific exceptions must come before general ones
- Only one catch block executes

Multiple catch blocks improve clarity and precise error handling.

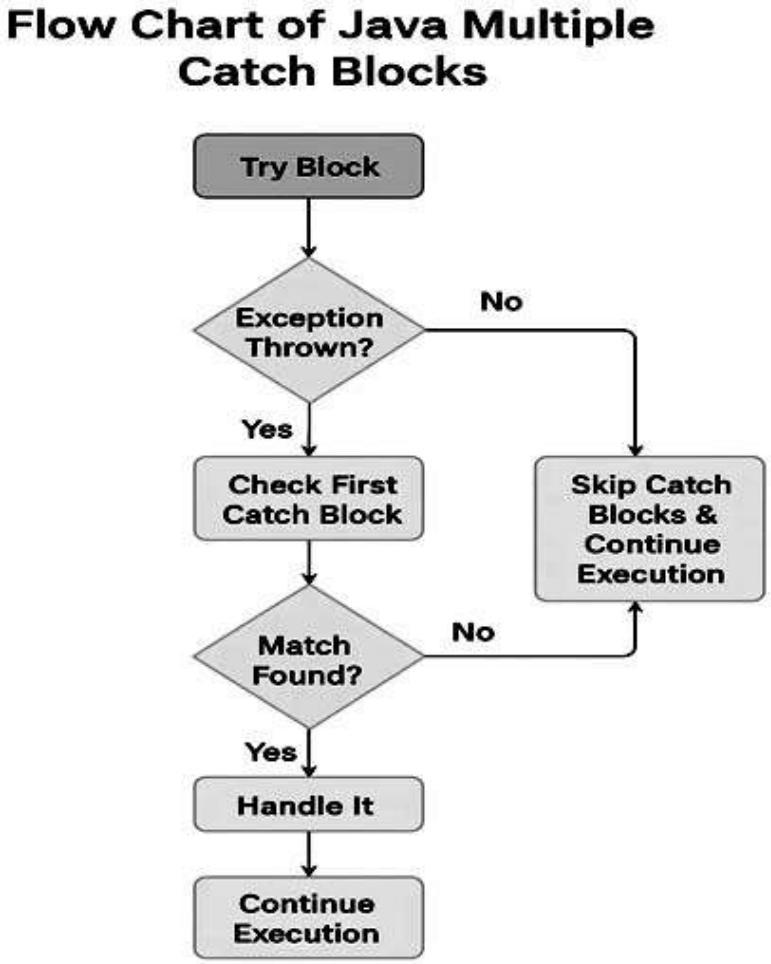


Fig. 19.6 : Java Multiple Catch Blocks.

NOTES

19.4.3. throw and throws

Java provides throw and throws keywords for explicit exception handling.

throw Keyword

Used to explicitly throw an exception.

throw new ArithmeticException ("Invalid operation");

throws Keyword

Used in method declaration to indicate possible exceptions.

```
void readFile() throws IOException {
    // code
}
```

Difference Between throw and throws

throw	throws
Used inside method	Used in method signature
Throws single exception	Declares multiple exceptions
Explicit	Implicit

These keywords provide better control over exception propagation.

Activity 19.1. Identify and Handle Errors

Objectives:

- Understand types of exceptions
- Develop problem-solving skills

Materials Needed:

- Notebook

Activity Description:
Think of a system (e.g., online payment). List 3–4 possible errors and suggest how each can be handled.

Reflection/Outcome:
Write 2–3 lines on why handling errors is important.

☑ Check Your Progress 19.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. What is an interface in Java? How is it defined and implemented?

. . . . . .

. . . . . .

. . . . . .

2. What are user-defined packages? Why are they used?

. . . . . .

. . . . . .

. . . . . .

3. Explain exception handling in Java with try–catch, multiple catch, and throw/throws.

. . . . . .

. . . . . .

. . . . . .

19.5. SUMMARY

This unit reinforces continuity between earlier Java concepts and upcoming advanced topics. Clearly defined objectives guide learners toward measurable learning outcomes. Emphasis is placed on logical reasoning and structured program development. The unit supports revision of foundational concepts to strengthen understanding. Academic tools such as glossary and exercises enhance self-learning. Self-assessment helps learners evaluate comprehension and progress. Suggested readings promote independent and extended learning. Overall, the unit strengthens the academic base required for advanced Java programming

19.6. GLOSSARY

- **Objective –** A clearly defined and measurable learning outcome
- **Concept Reinforcement –** Strengthening previously learned programming ideas
- **Syntax –** Rules that define the structure of Java statements
- **Logic –** Step-by-step reasoning used to solve programming problems
- **Compilation –** Process of converting Java source code into bytecode
- **Execution Flow –** Order in which program instructions are executed
- **Runtime –** Phase during which a program is running
- **Error –** A mistake that leads to incorrect program behavior
- **Debugging –** Process of identifying and correcting errors
- **Self-Learning Material (SLM) –** Structured content designed for independent study

NOTES

19.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** An interface in Java is a reference type that contains abstract methods (and constants). It is defined using the interfacekeyword. A class implements an interface using the implements keyword and must provide implementations for all its methods. Interfaces support multiple inheritance and help achieve abstraction.
2. **Answer :** User-defined packages are custom packages created by programmers to organize classes and interfaces into namespaces. They help in avoiding naming conflicts, improving code readability, and supporting reusability and modular programming.
3. **Answer :** Exception handling in Java is used to manage runtime errors and ensure smooth program execution.
   - The **try–catch block** is used to handle exceptions by placing risky code in the try block and handling errors in the catch block.
   - **Multiple catch** allows handling different types of exceptions separately.
   - The **throw** keyword is used to explicitly throw an exception, while **throws** is used in method declarations to indicate that a method may throw exceptions.
   - to assess understanding and application skills.

19.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3760	

19.9. REFERENCES

- Balagurusamy, E. (2022). *Programming with Java*. McGraw-Hill Education.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Oracle. (2023). *Java Documentation*.
- NPTEL. (2022). *Java Programming Courses*.

19.10 EXERCISE

Part 1

A. Multiple Choice Question

1. An interface in Java is:
   (a) A class with implementation
   (b) A collection of abstract methods
   (c) A loop
   (d) A variable
2. A class implements an interface using:
   (a) extends
   (b) implements
   (c) inherits
   (d) imports
3. Packages in Java are used to:
   (a) Execute programs
   (b) Organize classes and interfaces
   (c) Create loops
   (d) Define variables
4. The try–catch block is used for:
   (a) Looping
   (b) Handling exceptions
   (c) Declaring variables
   (d) Creating objects
5. The throw keyword is used to:
   (a) Handle exceptions
   (b) Create exceptions explicitly
   (c) Stop execution
   (d) Define methods

Answers with Explanations

1. Ans. (b)

Interface Concept : An interface is a collection of abstract methods used to achieve abstraction and multiple inheritance.

2. Ans. (b)

Implementing Interface : A class uses the implements keyword to implement an interface and provide method definitions.

3. Ans. (b)

Packages : Packages are used to organize related classes and interfaces, improving code structure and reusability.

4. Ans. (b)

Exception Handling : The try–catch block is used to handle runtime errors and prevent program crashes.

NOTES

5. Ans. (b)

throw Keyword : The throw keyword is used to explicitly create and throw an exception.

B. Short Answer Questions (50–100 words)

1. Explain the importance of objectives in an SLM unit.
2. Describe the role of a glossary in learning Java.
3. Why are summaries important in academic units?
4. What is the purpose of suggested readings?
5. Define debugging.

Part 2

C. Long Answer Questions (200 words)

1. Discuss how structured objectives support effective learning in Java.
2. Explain the role of exercises in self-learning materials.

Part 3

D. Long Answer Questions (300 words)

1. Explain how SLM structure supports independent learning in Java programming.
2. Discuss the importance of summaries, glossary, and exercises in reinforcing concepts.

UNIT – 20
MULTITHREADING, APPLETS, I/O AND GRAPHICS

UNIT STRUCTURE

20.1 Introduction
20.2 Objective
20.3 Multithreading and Applets
20.4 I/O Streams and Graphics
20.5 Summary
20.6 Glossary
20.7 Answer to check your progress
20.8 Suggested readings - OER
20.9 Reference
20.10 Exercise

20.1. INTRODUCTION

In the previous units, learners have studied advanced Java programming concepts such as interfaces, packages, exception handling, inheritance mechanisms, and abstraction. These concepts form a strong foundation for developing reliable, modular, and scalable Java applications. As student's progress further, it becomes important to consolidate this knowledge and prepare for more specialized and application-oriented topics in Java. Unit 20 is designed to strengthen conceptual clarity and encourage systematic learning. This unit emphasizes the importance of disciplined programming practices, structured thinking, and effective problem-solving strategies. By reinforcing previously learned concepts, learners are better equipped to understand how different components of Java programming interact in real-world software development. The self-learning material (SLM) approach adopted in this unit provides clear objectives, concise summaries, and academic support tools such as glossary, exercises, and suggested readings. These components help learners revise essential ideas, assess their understanding, and explore additional learning resources independently. The unit also promotes analytical thinking and logical reasoning, which are crucial for mastering Java programming.

By the end of Unit 20, learners are expected to demonstrate improved confidence in applying Java concepts, analyzing program behavior,

NOTES

and approaching programming challenges with a structured and methodical mindset. Thus, Unit 20 serves as an important academic link in the continuous learning journey of Java programming.

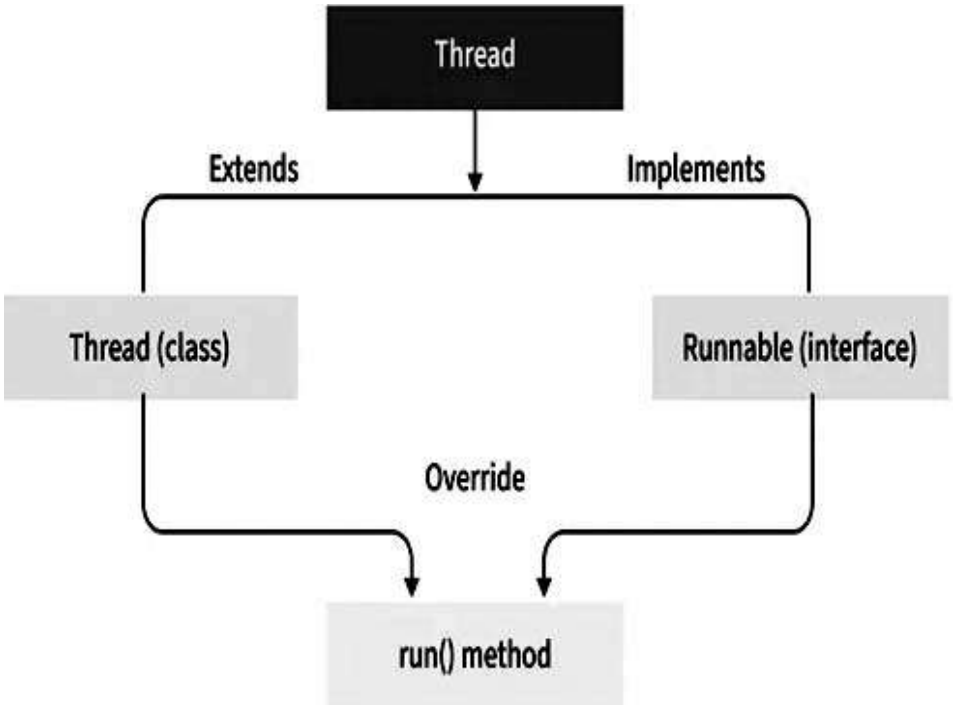
20.2. OBJECTIVES

After completing this unit, the learner will be able to:

- **Recall** key Java programming concepts studied in earlier units
- **Identify** important terminology and ideas related to this unit
- **Explain** the role of structured learning in Java programming
- **Apply** previously learned concepts in basic programming contexts
- **Analyze** program logic and execution flow
- **Develop** systematic and disciplined problem-solving skills

20.3. MULTITHREADING AND APPLETS

Modern Java applications often need to perform multiple tasks simultaneously, such as handling user input, performing background calculations, and managing system resources. Java supports such concurrent execution through multithreading. Alongside this, Java historically provided applets to create interactive, graphical programs that run inside a web browser. Understanding both concepts is essential for appreciating Java's capabilities in concurrency and graphical programming. Multithreading improves performance, responsiveness, and efficient CPU utilization, while applets introduced Java's ability to create platform-independent graphical programs. Although applets are now deprecated, they remain important from an academic and conceptual perspective.



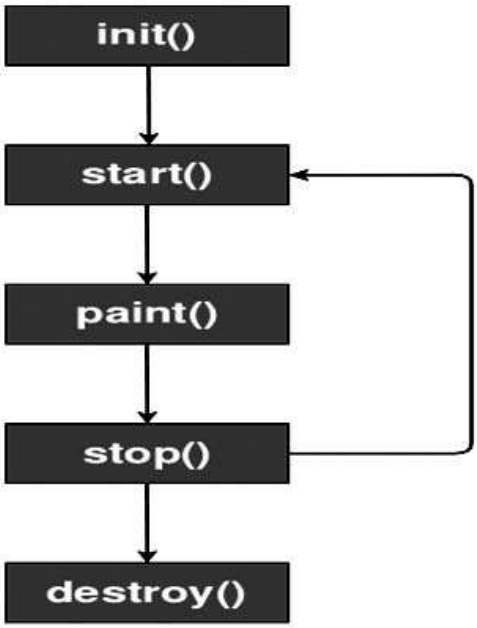


Fig. 20.1 : Multithreading And Applets.

20.3.1. Thread Life Cycle

A thread is a lightweight sub-process that represents an independent path of execution within a program. In Java, every thread goes through a series of states known as the thread life cycle. Understanding these states helps programmers design efficient and error-free multithreaded applications.

States of Thread Life Cycle

1. **New State**
   - A thread is in the new state when it is created but not yet started.
2. **Thread t = new Thread();**
3. **Runnable State**
   - The thread is ready to run and waiting for CPU time.
4. **t.start();**
5. **Running State**
   - The thread is currently executing.
6. **Blocked / Waiting State**
   - The thread is temporarily inactive due to resource waiting, sleep, or join.
7. **Terminated (Dead) State**
   - The thread completes execution or is stopped.

Importance of Thread Life Cycle

- Helps manage thread execution
- Avoids deadlocks and starvation
- Improves program efficiency

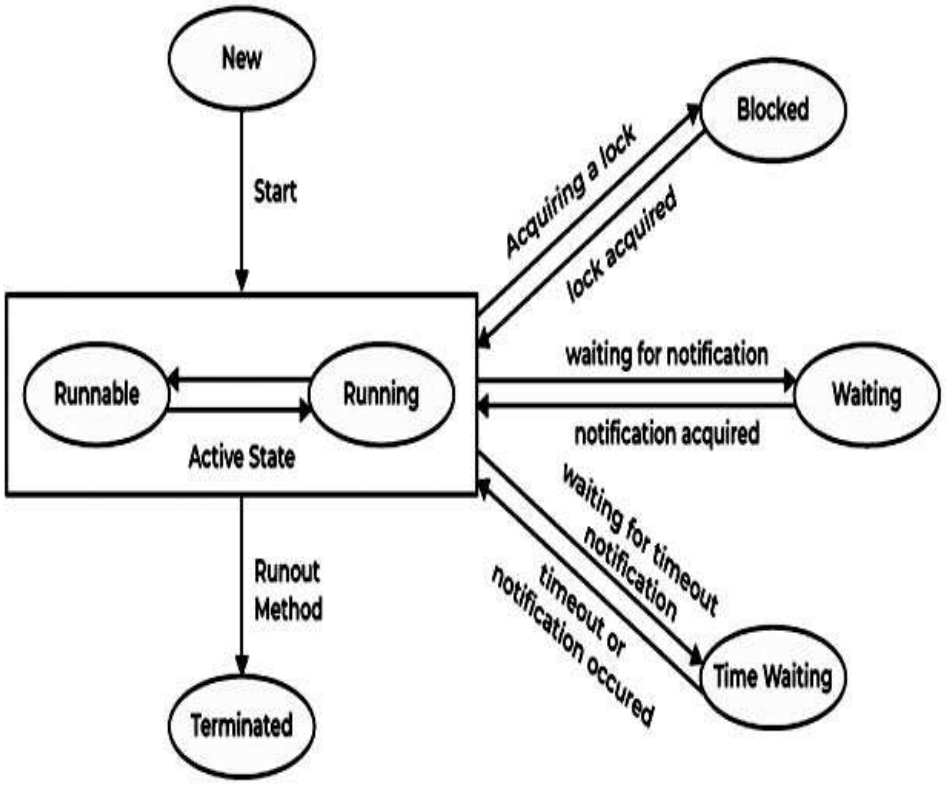


Fig. 20.2 : Multithreading And Applets.

20.3.2. Thread Methods

Java provides a rich set of thread methods to control thread execution.

Common Thread Methods

- **start() –** Starts thread execution
- **run() –** Defines thread task
- **sleep() –** Pauses thread for a specified time
- **join() –** Waits for another thread to finish
- **yield() –** Temporarily pauses current thread
- **isAlive() –** Checks if thread is active

Example

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running");
    }
}
```

Thread methods enable synchronization, coordination, and controlled execution in multithreaded programs.

20.3.3. Applet Life Cycle

An applet is a special type of Java program that is designed to run inside a web browser or an applet viewer. Unlike standard Java

applications, applets do not have a main() method. Instead, their execution is controlled by the browser or applet viewer, which follows a predefined sequence of method calls known as the Applet Life Cycle. Understanding the applet life cycle is essential for learning how Java manages graphical, event-driven programs.The applet life cycle defines the various stages through which an applet passes from creation to destruction. At each stage, a specific method is automatically invoked by the Java runtime environment. These methods allow the programmer to initialize resources, start execution, display graphics, pause execution, and release resources in a systematic manner. Although applets are now deprecated in modern Java, they remain an important academic topic because they demonstrate concepts such as event-driven programming, graphical rendering, and managed execution flow. The life cycle model of applets also helps students understand how Java-based components interact with their execution environment.

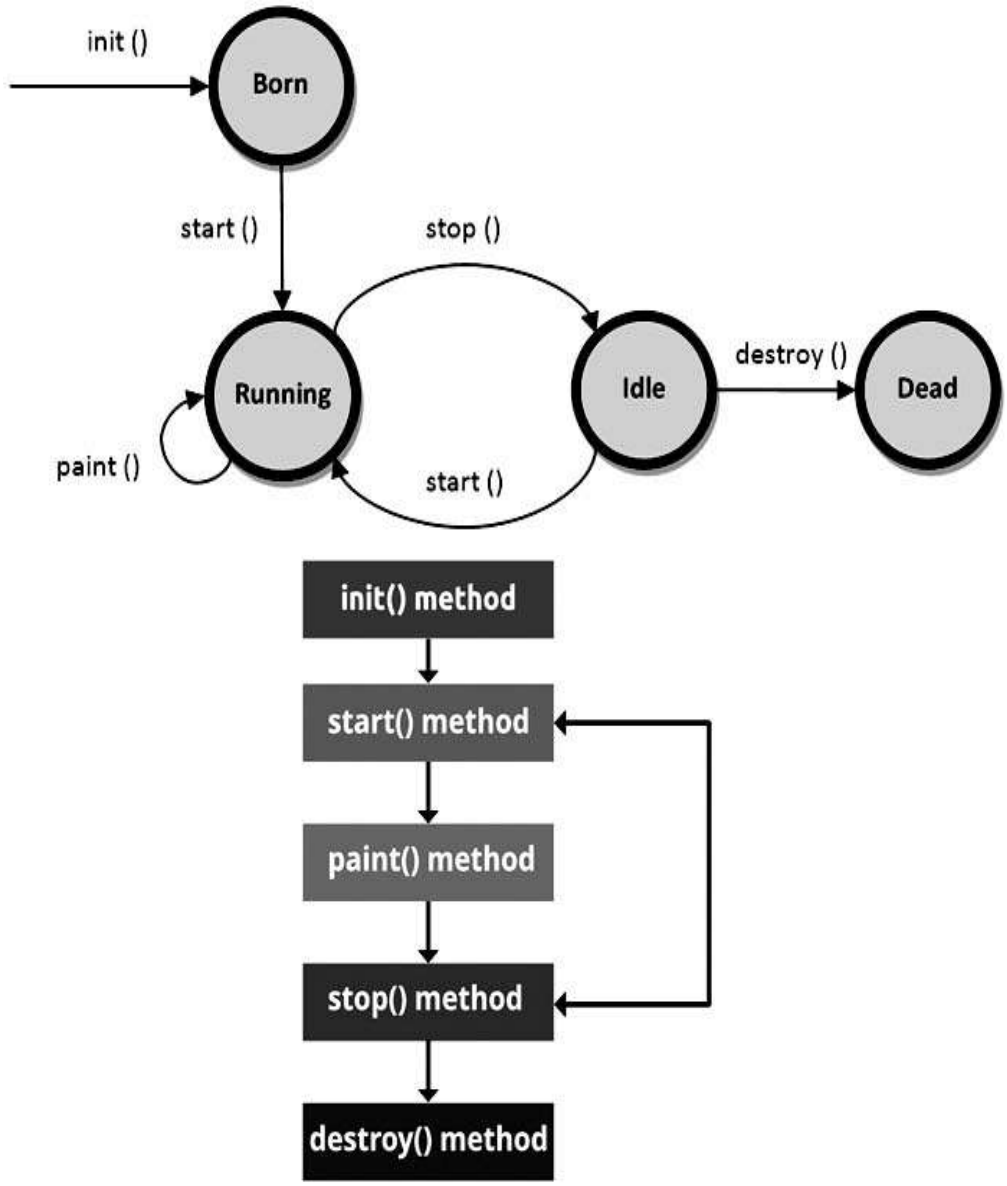


Fig. 20.3 : Applet Life Cycle.

NOTES

Overview of Applet Life Cycle

The applet life cycle consists of the following major methods:

1. init()
2. start()
3. paint()
4. stop()
5. destroy()

These methods are defined in the Applet class (in java.applet package) and are automatically called by the browser or applet viewer. The programmer overrides these methods to define the desired behavior of the applet at each stage.

1. init() Method

The init() method is the first method invoked in the life cycle of an applet. It is called only once, when the applet is loaded into memory.

Purpose of init()

- Initialize variables
- Allocate resources
- Set up user interface components
- Load images, fonts, or other data

Characteristics

- Called automatically
- Executed only once
- Used for initialization tasks

Example

```
publicvoidinit() {
    setBackground(Color.cyan);
}
```

The init() method plays a role similar to a constructor, but it is controlled by the browser rather than the programmer.

2. start() Method

The start() method is called after init() and every time the applet becomes active. This may occur when:

- The applet is first displayed
- The user revisits the webpage containing the applet

Purpose of start()

- Start animations
- Start threads

- Resume execution

Characteristics

- Can be called multiple times
- Executed after init()
- Used to begin or resume activity

Example

```
publicvoidstart() {
    System.out.println("Applet started");
}
```

The start() method ensures that the applet responds dynamically whenever it becomes visible or active.

3. paint() Method

The paint() method is responsible for displaying output on the screen. It is automatically invoked whenever the applet needs to redraw its content, such as:

- When the applet is first displayed
- When the window is resized
- When repaint() is called

Purpose of paint()

- Draw text
- Draw shapes
- Display images

Syntax

```
publicvoidpaint(Graphics g) {
    g.drawString("Hello Applet", 50, 50);
}
```

Characteristics

- Receives a Graphics object
- Called multiple times
- Used only for drawing

The paint() method is central to graphical programming in Java, as it controls how visual elements appear on the screen.

4. stop() Method

The stop() method is called when the applet becomes inactive. This happens when:

- The user navigates away from the webpage
- The browser window is minimized

NOTES

Purpose of stop()

- Pause animations
- Suspend threads
- Temporarily stop execution

Characteristics

- Can be called multiple times
- Used to release temporary resources
- Helps conserve system resources

Example

```
publicvoidstop() {
    System.out.println("Applet stopped");
}
```

The stop() method ensures that unnecessary processing does not continue when the applet is not visible.

5. destroy() Method

The destroy() method is the last method in the applet life cycle. It is called when the applet is about to be removed permanently from memory.

Purpose of destroy()

- Release system resources
- Close files or connections
- Perform final cleanup

Characteristics

- Called only once
- Executed before applet termination
- Used for cleanup operations

Example

```
publicvoiddestroy() {
    System.out.println("Applet destroyed");
}
```

This method is like a cleanup or termination phase in application life cycles.

Applet Life Cycle Execution Flow

The general execution order is:

1. init() → called once
2. start() → called when applet becomes active
3. paint() → called whenever display is needed
4. stop() → called when applet becomes inactive

5. destroy() → called before removal

This structured execution ensures controlled initialization, execution, and termination of applets.

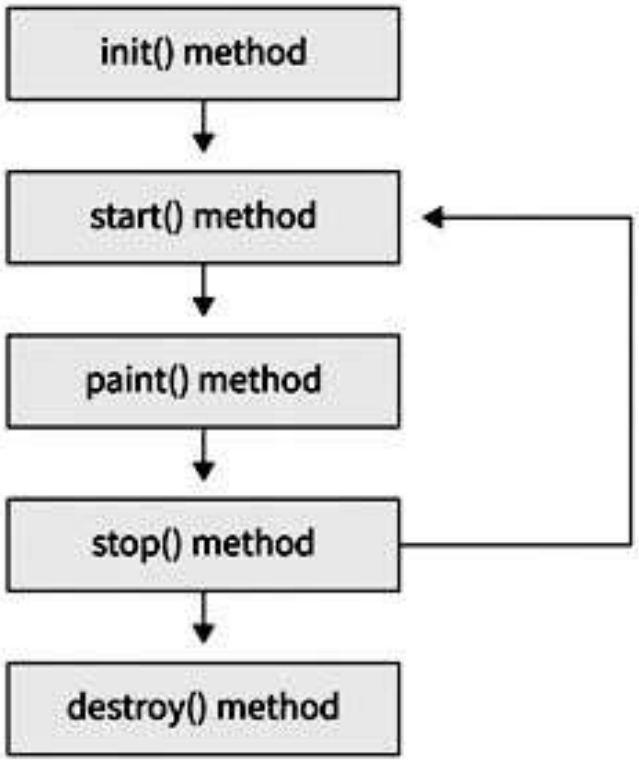


Fig. 20.4 : Cycle Execution Flow.

Advantages of Applet Life Cycle

- Provides structured execution
- Automatically managed by browser
- Supports graphical and event-driven programming
- Simplifies resource management

Limitations of Applets

- Security restrictions
- Browser dependency
- Deprecated in modern Java
- Limited real-world usage today
- Despite these limitations, applets remain important for understanding Java GUI concepts and managed execution models.

20.4. I/O STREAMS AND GRAPHICS

Java applications frequently need to read input, write output, store data, and display graphical content. Java provides I/O streams for data handling and the Graphics class for drawing shapes and images. These features are crucial for file processing and GUI-based applications.

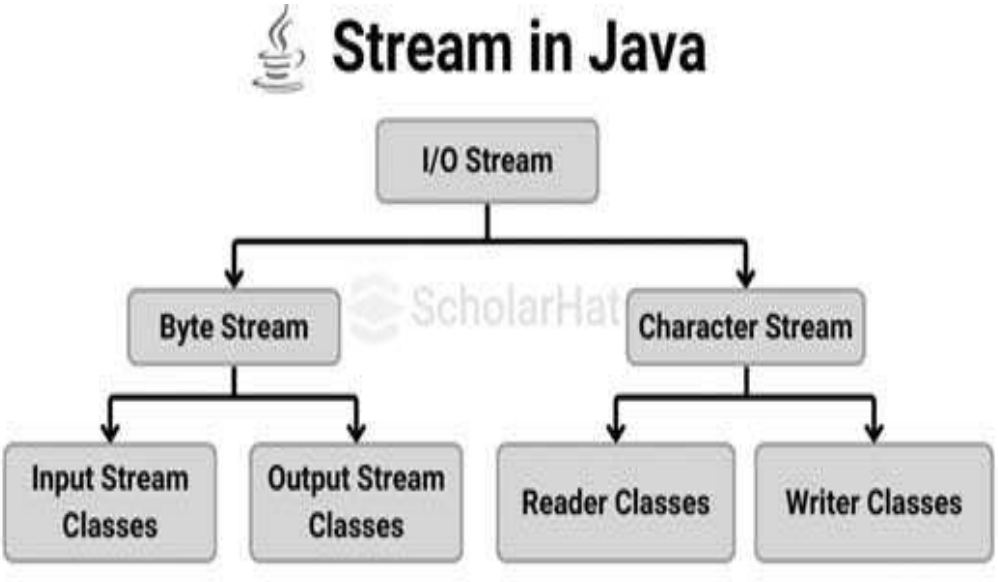


Fig. 20.5 : Stream in Java.

20.4.1. Byte and Character Streams

Java I/O streams are categorized into byte streams and character streams.

Byte Streams

- Handle binary data
- Use InputStream and OutputStream

FileInputStream fis = new FileInputStream("data.txt");

Character Streams

- Handle text data
- Use Reader and Writer

FileReader fr = new FileReader("data.txt");

Differences

Byte Streams	Character Streams
Binary data	Text data
Faster	Unicode support

20.4.2. File Handling

File handling allows Java programs to store and retrieve data from files.

Common File Classes

- File
- FileInputStream
- FileOutputStream
- FileReader
- FileWriter

Example

```
File f = new File("example.txt");
```

Advantages

- Persistent data storage
- Easy data retrieval
- Supports large applications

20.4.3. Graphics Class and Drawing

The Graphics class in Java is used to draw shapes, text, and images in GUI-based programs and applets.

Common Drawing Methods

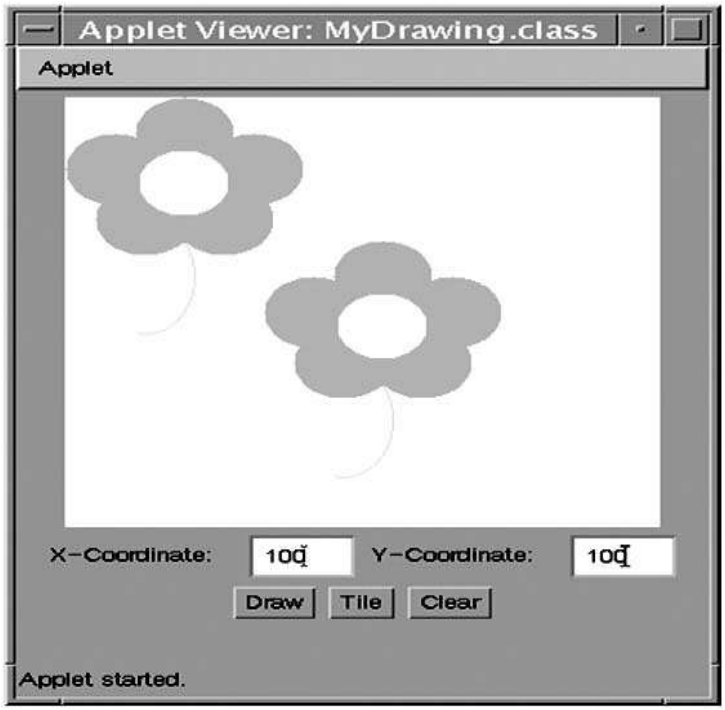
- drawLine()
- drawRect()
- drawOval()
- drawString()

Example

```
public void paint(Graphics g)
{
    g.drawRect(50, 50, 100, 100);
}
```

Applications

- Drawing charts
- Game development
- GUI components



NOTES

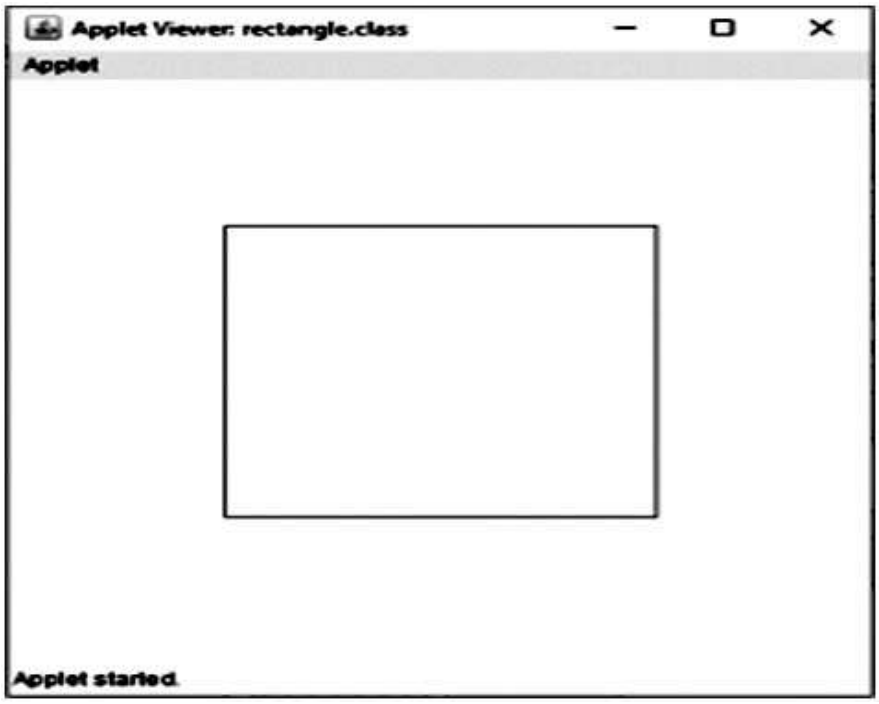


Fig. 20.6 : Graphics Class and Drawing.

Activity 20.1.

Develop a simple multithreading program. Experiment with basic file input/output operations or simple graphics if possible.

☑ Check Your Progress 20.1.

Notes : (a) Space is given below for your answer.

(b) Check your answer with the one given at the end of this unit.

1. Explain the thread life cycle and common thread methods in Java.

. . . . . .

. . . . . .

. . . . . .

2. What is the applet life cycle? Describe its main stages.

. . . . . .

. . . . . .

. . . . . .

3. What are I/O streams in Java? Explain byte streams, character streams, and the role of the Graphics class.

. . . . . .

. . . . . .

. . . . . .

20.5. SUMMARY

- This unit reinforces continuity between previously learned Java concepts and upcoming advanced topics
- Clearly defined objectives guide learners toward measurable learning outcomes
- Emphasis is placed on logical reasoning and structured program development
- The unit supports revision of foundational concepts to strengthen understanding
- Academic tools such as glossary and exercises enhance self-learning
- Self-assessment helps learners evaluate comprehension and progress
- Suggested readings encourage independent and extended learning
- Overall, the unit strengthens the academic base required for advanced Java programming

20.6. GLOSSARY

- **Objective** – A clearly defined and measurable learning outcome
- **Concept Reinforcement** – Strengthening previously learned programming ideas
- **Syntax** – Rules that define the structure of Java statements
- **Logic** – Step-by-step reasoning used to solve programming problems
- **Compilation** – Process of converting Java source code into bytecode
- **Execution Flow** – Order in which program instructions are executed
- **Runtime** – Phase during which a program is running
- **Error** – A mistake that leads to incorrect program behavior
- **Debugging** – Process of identifying and correcting program errors
- **Self-Learning Material (SLM)** – Structured content designed for independent study

20.7. ANSWERS TO CHECK YOUR PROGRESS

1. **Answer :** The thread life cycle in Java includes several stages: New, Runnable, Running, Blocked/Waiting, and Terminated. A thread starts in the new state, becomes runnable, executes in the running state, may enter waiting or blocked state, and finally terminates. Common thread methods include start(), run(), sleep(), join(), and yield(), which control thread execution.

NOTES

2. Answer : The applet life cycle consists of the following stages:

- init() – Initializes the applet
- start() – Starts or resumes execution
- stop() – Pauses execution
- destroy() – Cleans up resources before termination
- paint() – Used for rendering output on the screen

3. Answer : I/O streams in Java are used for input and output operations.

- Byte streams handle binary data (e.g., images, files) using classes like InputStream and OutputStream.
- Character streams handle text data using Reader and Writer classes.
- File handling allows reading from and writing to files.
- The Graphics class is used for drawing shapes, text, and images in Java applications and applets.

20.8. SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource	http://books.aisectonline.com/User/UnitUrls?unitId=3761	

20.9. REFERENCES

- Balagurusamy, E. (2022). *Programming with Java*. McGraw-Hill Education.
- Schildt, H. (2021). *Java: The Complete Reference*. McGraw-Hill.
- Oracle. (2023). *Java Documentation*.
- NPTEL. (2022). *Java Programming Courses*.

20.10. EXERCISE

Part 1

A. Multiple Choice Question

1. Multithreading in Java is used to:
   (a) Execute one task at a time
   (b) Execute multiple tasks simultaneously
   (c) Store data
   (d) Define classes

2. The thread life cycle includes:
   (a) Only start and stop
   (b) New, Runnable, Running, Waiting, Terminated
   (c) Compile and run
   (d) Input and output
3. Which method is used to start a thread?
   (a) run()
   (b) start()
   (c) execute()
   (d) begin()
4. The applet life cycle includes:
   (a) init(), start(), stop(), destroy()
   (b) main(), run(), stop()
   (c) create(), execute()
   (d) input(), output()
5. Byte streams are used to:
   (a) Handle only text data
   (b) Handle binary data
   (c) Execute programs
   (d) Create threads

Answers with Explanations

1. Ans. (b)

Multithreading : Multithreading allows multiple tasks to run concurrently, improving performance and efficiency.

2. Ans. (b)

Thread Life Cycle : A thread goes through stages like New, Runnable, Running, Waiting/Blocked, and Terminated.

3. Ans. (b)

Thread Start Method : The start() method is used to begin the execution of a thread.

4. Ans. (a)

Applet Life Cycle : Applet execution follows stages such as init(), start(), stop(), and destroy().

5. Ans. (b)

Byte Streams : Byte streams are used to handle binary data such as images, audio, and files.

NOTES

B. Short Answer Questions (50–100 words)

1. What is multithreading in Java?
2. Explain the thread life cycle.
3. What are thread methods?
4. Describe the applet life cycle.
5. What are byte and character streams?

Part 2

C. Long Answer Questions (200 words)

1. Explain multithreading and thread life cycle in Java.
2. Discuss I/O streams and file handling in Java.

D. Long Answer Questions (~300 words)

1. Explain thread methods and applet life cycle in detail.
2. Discuss byte streams, character streams, and the Graphics class with examples.

DR. C.V. RAMAN UNIVERSITY

Chhattisgarh, Bilaspur AN AISECT GROUP UNIVERSITY

Approved by : PCI | AICTE | NCTE | BCI | Member of : AIU | Joint Committee : (UGC | DEB | AICTE) | Recognized by : UGC | A NAAC Accredited University

DR. C. V. RAMAN UNIVERSITY (C.G.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org/ for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***