

OBJECT ORIENTED PROGRAMMING WITH JAVA

B.Com., COMPUTER APPLICATION / BA. COMPUTER APPLICATION

Semester – IV

Lesson Writers

Dr. Kampa Lavanya

Asst. professor

Dept. of Computer Science

Acharya Nagarjuna University

Dr. U. Surya Kameswari

Asst. professor

Dept. of Computer Science

Acharya Nagarjuna University

Editor

Dr. KAMPA LAVANYA

Asst. Professor

Department of Computer Science

Acharya Nagarjuna University,

Director I/c

Prof. V.VENKATESWARLU

MA., M.P.S., M.S.W., M.Phil., Ph.D.

CENTRE FOR DISTANCE EDUCATION

ACHARAYANAGARJUNAUNIVERSITY

NAGARJUNANAGAR – 522510

Ph:0863-2346222,2346208,

0863-2346259(Study Material)

Website: www.anucde.info

e-mail:anucdedirector@gmail.com

B.A./ B.Com –OBJECT ORIENTED PROGRAMMING WITH JAVA

First Edition: 2021

No. of Copies:

©Acharya Nagarjuna University

**This book is exclusively prepared for the use of students of B.A/ B.Com., (Computer Application)
Centre for Distance Education, Acharya Nagarjuna University and this book is meant for limited
Circulation only.**

Published by:

**Prof. V.VENKATESWARLU,
Director I/c,
Centre for Distance Education,
Acharya Nagarjuna University**

Printed at:

FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A' grade from the NAAC in the year 2016, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 443 affiliated colleges spread over the two districts of Guntur and Prakasam.

The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavours.

*Prof. K. Gangadhara Rao
Vice-Chancellor I/c
Acharya Nagarjuna University*

B.Com., COMPUTER APPLICATION / BA. COMPUTER APPLICATION
Semester – IV
409BCO21: OBJECT ORIENTED PROGRAMMING WITH JAVA
SYLLABUS

Unit: 1

Introduction to OOPs: Problems in Procedure Oriented Approach, Features of Object Oriented Programming

Introduction to Java: Features of Java, The Java Virtual Machine (JVM), Parts of Java program, Naming Conventions in Java, Data Types in Java, Operators in Java, Reading Input using scanner Class, Displaying Output using System. out. Print In (), Command Line Arguments.

Unit II:

Control Statements in Java: if... else, do... while Loop, while Loop, For loop, Switch Statement, break Statement, continue Statement

Arrays: Types of Arrays, array name, length,

Strings: Creating Strings, String Class Methods, String Comparison, Immutability of Strings.

Unit III:

Classes and Objects: Object Creation, Initializing the Instance Variables, Access Specifiers, Constructors

Inheritance: Inheritance, Types of Inheritance

Polymorphism: Method overloading, Operator overloading

Abstract Classes: Abstract Method and Abstract Class

Unit IV:

Packages: Package, Different Types of Packages, Creating Package and Accessing a Package

Streams: Stream classes, Creating a File using File Output Stream, Reading Data from a File using File Input Stream, Creating a File using File Writer, Reading a File using File Reader

Unit V:

Exception Handling: Errors in Java Program, Exceptions, throws Clause, throw Clause, Types of Exceptions

Threads: Single Tasking, Multi-Tasking, Uses of Threads, Creating a Thread and Running it, Terminating the Thread, Thread Class Methods.

REFERENCES:

1. The Complete Reference JAVA Seventh Edition Herbert Schildt. Tata McGraw Hill Edition.
2. Core Java: An Integrated Approach, Dr. R. Nageswara Rao & Kogent Learning Solutions Inc.
3. E. Balaguruswamy, Programming with JAVA, A primer, 3e, TATA McGrawHill Company

ONLINE RESOURCES:

<https://stackify.com/java-tutorials/>

<https://www.w3schools.com/java/>

<https://www.javatpoint.com/java-tutorial>

<https://www.tutorialspoint.com/java/index.html>

MODEL QUESTION PAPER

(409BCO21)

B.A./B.Com. (Computer Applications) DEGREE EXAMINATION

Fourth Semester

OBJECT ORIENTED PROGRAMMING WITH JAVA

Time : Three hours

Maximum : 70 marks

SECTION A — (5 × 4 = 20 marks)

Answer any FIVE of the following questions.

1. List the applications of Object-Oriented Programming
2. Reading input using Scanner Class.
3. Differentiate between break and continue statement.
4. Differentiate between a class and object.
5. What is the use of super keyword and 'this' keyword.
6. How to create and use a package in Java program?
7. Explain method overriding with a suitable example program.
8. How to assign priorities to threads?

SECTION B — (5 × 10 = 50 marks)

Answer the following questions.

9. (a) Explain the parts of Java program?
Or
(b) What are the data types and operators in Java? Give examples.
10. (a) What are the Control Statements in Java? Explain with suitable examples?
Or
(b) Write a program which stores a list of strings in an ArrayList and then displays the contents of the list.
11. (a) What is a nested class? Differentiate between static nested classes and non-static nested classes.
Or
(b) What is inheritance? Explain different forms of inheritance with suitable program.
12. (a) How to access, import a package? Explain with examples.
Or
(b) Explain the file management using File class.
13. (a) What are the uses of 'throw' and 'throws' clauses for exception handling?
Or
(b) Describe how to create a thread with an example.

CONTENTS

S.NO.	LESSON NAME	PAGES
1.	Introduction to OOP	1.1 – 1.12
2.	Introduction to JAVA	2.1 – 2.19
3.	Conditional Statements	3.1 – 3.16
4.	LOOP Statements	4.1 – 4.13
5.	ARRAYS and STRINGS	5.1 – 5.13
6.	CLASSES and OBJECTS	6.1 – 6.21
7.	Inheritance	7.1 – 7.16
8.	Polymorphism	8.1 – 8.16
9.	Packages	9.1 – 9.9
10.	Files	10.1 – 10.11
11.	Exceptional Handling	11.1 – 11.16
12.	Threads	12.1 – 12.10

LESSON- 1

INTRODUCTION TO OOP

OBJECTIVES

By the end of this chapter, you should be able to:

- Understand the basic concepts of Object-Oriented Programming (OOP).
- Identify the key differences between Procedure-Oriented Programming (POP) and OOP.
- Recognize the problems associated with Procedure-Oriented Programming.
- Explain the fundamental features of OOP, including encapsulation, inheritance, polymorphism, and abstraction.
- Apply the principles of OOP to design and implement modular and maintainable software solutions.

STRUCTURE

- 1.1 Introduction
- 1.2 Object-Oriented Programming (OOP)
 - 1.2.1 Key Concepts of OOP
 - 1.2.2 Benefits of OOP
- 1.3 Procedure Oriented Approach (POA)
 - 1.3.1 Key Characteristics of Procedure-Oriented Approach
 - 1.3.2 Problems with the Procedure-Oriented Approach
- 1.4 Features of OOP
- 1.5 Procedural Oriented Programming vs Object-Oriented Programming
- 1.6 Applications of OOP
- 1.7 Summary
- 1.8 Technical Terms
- 1.9 Self-Assessment Questions
- 1.10 Suggested Readings

1.1. INTRODUCTION

In the world of software development, different programming paradigms have been used to solve problems. The two most prominent paradigms are Procedure-Oriented Programming (POP) and Object-Oriented Programming (OOP). While POP was widely used in the early days of programming, it presented challenges in managing complex software projects.

This chapter introduces OOP, a paradigm that overcomes the limitations of POP by organizing software design around data, or objects, rather than functions and logic. You will learn how OOP offers a more intuitive approach to problem-solving by closely mirroring real-world entities and their interactions. Explain the fundamental features of OOP. Apply the principles of OOP to design and implement modular and maintainable software solutions.

1.2 OBJECT-ORIENTED PROGRAMMING (OOP)

is a programming paradigm that organizes software design around data, or objects, rather than functions and logic. An object in OOP is a self-contained unit that contains both

data (attributes) and methods (functions) that manipulate the data. OOP focuses on creating reusable code and models real-world entities and their interactions more naturally.

1.2.1 Key Concepts of OOP

The key concepts of Object-Oriented Programming (OOP) are essential principles that guide the design and implementation of software in an object-oriented way. The Key Concepts of OOP described in Figure 1.1.

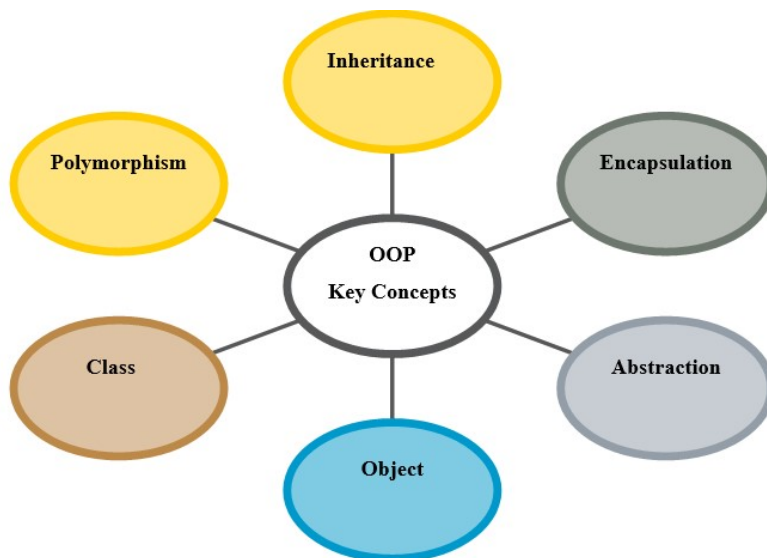


Fig 1.1. Key Concepts of OOP

❖ Object

An object is an instance of a class. It represents a specific implementation of the class with actual values for its attributes. For example, a Car object might have make set to "Toyota", model set to "Corolla", and year set to 2021. Each object can interact with other objects or function independently.

Objects in OOP are fundamental to building complex systems, as they allow for encapsulation of data and behavior, promoting modularity and reuse. The concept is shown in Figure 1.2.

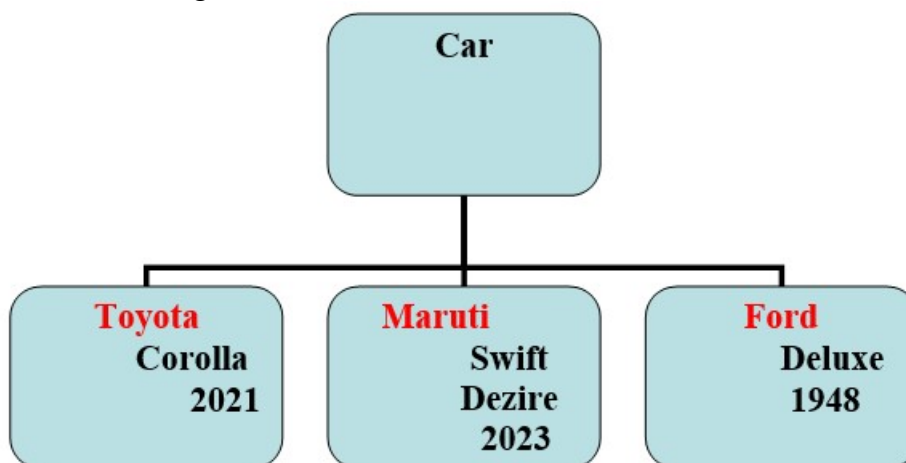


Fig 1.2. Object concept in OOP

❖ **Class**

A class is a blueprint for creating objects. It defines a data structure that holds attributes (data) and methods (functions) that manipulate this data. Classes allow programmers to create objects with specific properties and behaviors, providing a template that ensures consistency across similar objects.

For example, consider a class Car. The Car class might have attributes like make, model, and year, and methods like startEngine and stopEngine.

Every car object created from the Car class will have these attributes and methods, but with different values. The concept is shown in Figure 1.3.

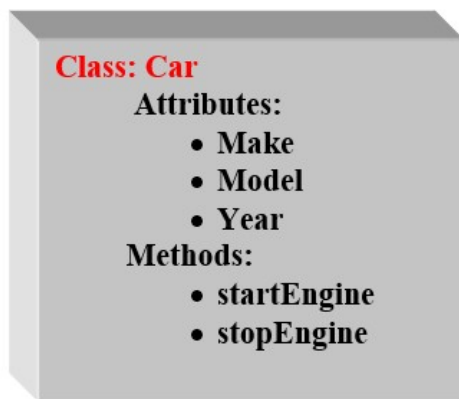


Fig 1.3. Class concept in OOP

❖ **Encapsulation:**

Encapsulation is the bundling of data and methods that operate on that data within a single unit, usually a class. It restricts access to certain components, which is essential for protecting the integrity of the data. By providing public methods to access private data, encapsulation enables controlled interaction with the data.

❖ **Inheritance:**

Inheritance is a mechanism that allows a new class, known as a subclass, to inherit attributes and methods from an existing class, known as a superclass. This promotes code reuse and establishes a natural hierarchy among classes. For example, a Circle and Box might inherit from the Shape class, sharing common attributes and methods while introducing new ones Circle and Box. The concept is shown in Figure 1.4.

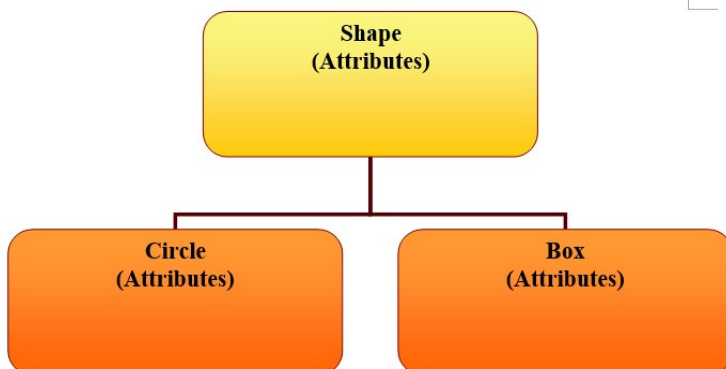


Fig 1.4. Inheritance concept in OOP

❖ Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying data types. For instance, both the Shape and Box classes might implement a Draw method, but each class could have a different implementation of this method. The concept is shown in Figure 1.5.

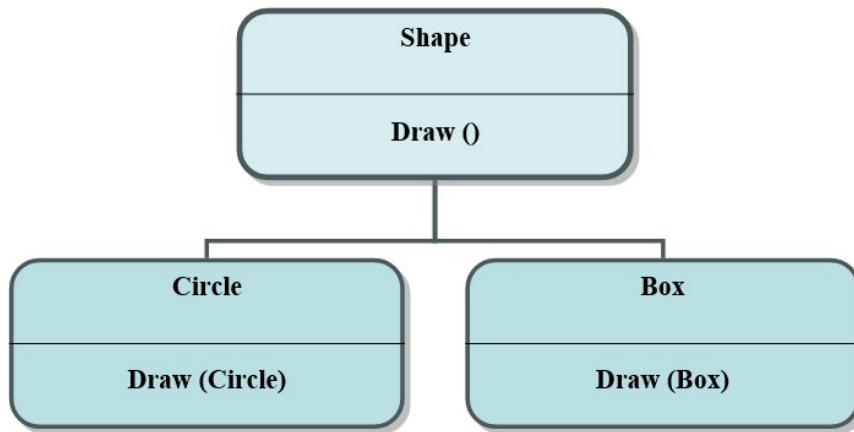


Fig 1.5. Polymorphism Concept in OOP

❖ Abstraction:

Abstraction involves hiding the complex implementation details of a class and exposing only the necessary interfaces to the user. It simplifies interaction with complex systems by focusing on the essential features and ignoring irrelevant details.

1.2.2 Benefits of OOP:

Object-Oriented Programming (OOP) offers several benefits that make it a preferred approach in modern software development. Here are the key benefits of OOP followed and described in figure 1.6:

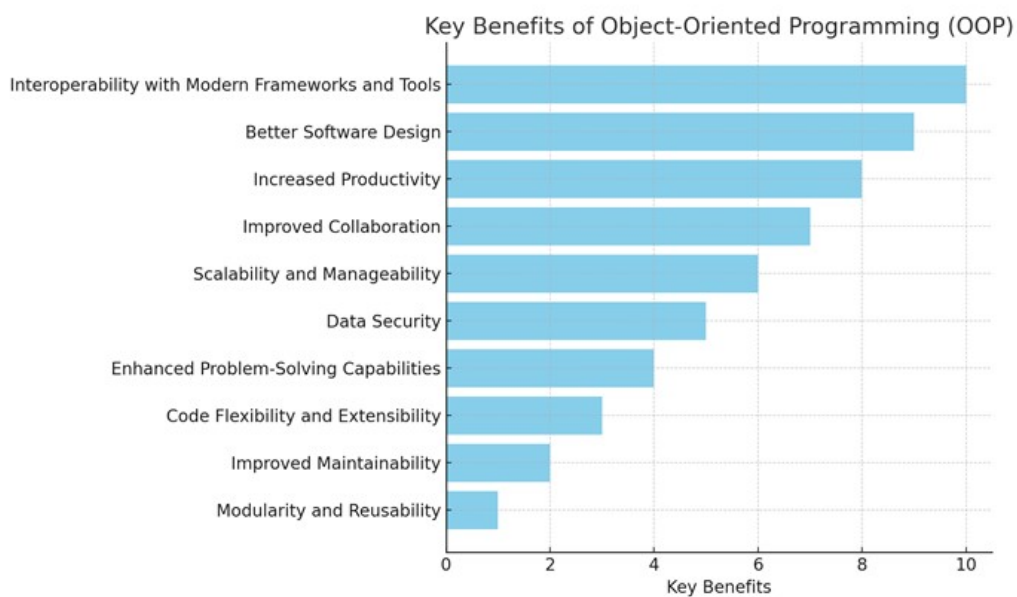


Fig 1.6 Key benefits of OOP

❖ **Modularity and Reusability**

- **Modularity:** OOP encourages the development of modular software, where the program is divided into discrete, manageable units or classes. Each class encapsulates data and behavior, making it easier to manage and understand.
- **Reusability:** Classes can be reused across different programs, reducing redundancy. Inheritance further promotes code reuse by allowing new classes to be created based on existing ones, minimizing the need to write code from scratch.

❖ **Improved Maintainability**

- **Ease of Maintenance:** The modular nature of OOP makes it easier to update or modify parts of a program without affecting the entire system. Since changes are often localized within specific classes, maintaining the software becomes more straightforward and less error-prone.
- **Encapsulation:** By encapsulating data and exposing only necessary interfaces, OOP helps maintain the integrity of the data and reduces the risk of unintended side effects when making changes.

❖ **Code Flexibility and Extensibility**

- **Polymorphism:** OOP allows methods to be overridden or objects to be treated as instances of their superclass, enabling flexible and dynamic code. This allows developers to write more generic and adaptable code that can work with different types of objects.
- **Inheritance:** New functionality can be added to existing classes through inheritance, allowing developers to extend or modify behavior without changing the existing codebase. This makes it easier to scale applications as new requirements emerge.

❖ **Problem-Solving Capabilities**

- **Real-World Modeling:** OOP aligns closely with real-world concepts, making it easier to model complex systems and problems. By representing real-world entities as objects with attributes and behaviors, OOP provides an intuitive way to design and implement software.
- **Abstraction:** OOP allows developers to focus on high-level problem-solving by abstracting away complex details. This simplifies the design process and allows programmers to concentrate on the essential aspects of the problem.

❖ **Data Security**

- **Encapsulation:** Encapsulation helps protect data by restricting access to it. By controlling how data is accessed and modified, OOP ensures that the internal state of an object remains consistent and secure.
- **Access Control:** OOP provides mechanisms to define access levels (e.g., public, private, protected) for class members, ensuring that sensitive data is not exposed or altered inappropriately.

❖ **Scalability and Manageability**

- **Scalability:** OOP systems are inherently scalable due to their modular design. New features can be added with minimal impact on existing code, making it easier to scale applications as they grow in size and complexity.
- **Manageability:** The clear structure of OOP programs, with distinct classes and well-defined interfaces, makes large codebases easier to manage, understand, and document.

❖ Improved Collaboration

- **Team Development:** OOP's modular approach allows different team members to work on separate classes or modules simultaneously, improving collaboration and speeding up development.
- **Code Sharing:** Reusable classes and components can be shared across teams or projects, fostering collaboration and reducing duplication of effort.

❖ Increased Productivity

- **Rapid Development:** OOP's features like inheritance, polymorphism, and reusable components can significantly speed up the development process. Developers can build upon existing code rather than starting from scratch, leading to faster and more efficient coding.
- **Code Reuse:** The ability to reuse classes across different projects or parts of a program reduces development time and increases productivity.

❖ Better Software Design

- **Design Patterns:** OOP encourages the use of design patterns, which are proven solutions to common design problems. These patterns help in creating robust, scalable, and maintainable software architectures.
- **Clear Structure:** OOP provides a clear and logical structure for software design, making it easier to plan, develop, and understand complex systems.

❖ Interoperability with Modern Frameworks and Tools

- **Support for Frameworks:** Many modern software development frameworks and libraries are designed with OOP principles in mind. Using OOP makes it easier to integrate with these tools and take advantage of their capabilities.
- **Adaptation to New Technologies:** OOP principles are widely adopted in various programming languages, making it easier to adapt to new technologies, platforms, and languages that also support OOP.

These benefits make OOP a powerful and effective approach for developing complex, scalable, and maintainable software systems.

1.3 PROCEDURE ORIENTED APPROACH (POA)

The Procedure-Oriented Approach (also known as Procedural Programming) is a programming paradigm that is centred around the concept of procedure calls, where procedures, also known as routines, subroutines, or functions, are the fundamental building blocks. This approach is one of the earliest and most widely used paradigms, especially in languages like C, Pascal, and Fortran.

1.3.1 Key Characteristics of Procedure-Oriented Approach:

The Procedure-Oriented Approach is a programming paradigm that centers around the concept of organizing code into procedures or functions. This approach is widely used in languages like C, Pascal, and Fortran. Below are the key characteristics of the Procedure-Oriented Approach and also concept is illustrated in figure 1.7:

❖ Focus on Functions

- The core idea of procedural programming is to structure the program as a series of functions or procedures, each designed to perform a specific task. Functions are the

building blocks of the program, and the overall program is a sequence of these function calls.

- **Example:** In a program that calculates the area of a rectangle, functions might include get Length (), get Width (), and calculate Area ().

❖ **Sequential Execution**

- The execution flow in a procedural program typically follows a linear and sequential order. The program starts from a specific entry point (often the main function) and proceeds through a series of function calls in a defined sequence.
- **Example:** A procedural program might execute functions in the order they are called within the main function, one after the other.

❖ **Global Data Sharing**

- Data in procedural programming is often stored in global variables that are accessible by multiple functions. These global variables are shared across different parts of the program, making it easier for functions to operate on the same data.
- **Example:** A global variable counter might be used and modified by various functions to track the number of times an operation is performed.

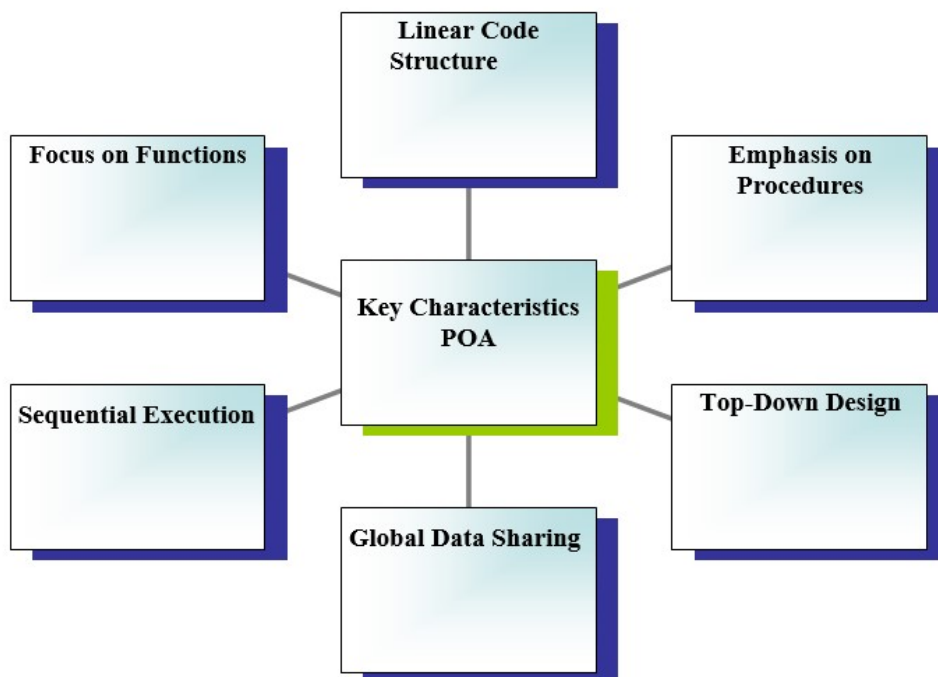


Fig 1.7. Key Characteristics of POA

❖ **Top-Down Design Approach**

- Procedural programming often follows a top-down design methodology, where the main problem is broken down into smaller, more manageable sub-problems. Each sub-problem is then solved by a specific function.
- **Example:** In designing a program to manage a library system, the top-down approach would first identify major functions like issue Book (), return Book (), and search Catalog (), and then further break these down into smaller functions.

❖ **Emphasis on Procedures Over Data**

- The primary focus in the procedural approach is on the procedures or functions themselves rather than the data being processed. The functions dictate how the data is manipulated and processed.
- **Example:** In a payroll system, the focus might be on functions like calculate Salary () and generate Pay slip () rather than on the employee data being processed.

❖ **Limited Modularity**

- While procedural programming allows for some degree of modularity by dividing the program into functions, this modularity is limited compared to Object-Oriented Programming (OOP). Functions are not encapsulated within objects and often depend on global data, reducing their independence.
- **Example:** Functions in a procedural program can be reused, but because they rely on global variables, their reusability is limited to contexts where those global variables are applicable.

❖ **Reusability**

- Functions can be reused within the program, especially if they are general-purpose. However, the reusability is not as robust as in OOP, where classes and objects can be more easily reused across different programs.
- **Example:** A function that calculates the sum of an array can be reused in different parts of the program, but it might need to be rewritten for a different context where the array format or data type changes.

❖ **No Data Hiding**

- In procedural programming, there is no inherent mechanism for hiding data from other parts of the program. Data is often accessible to any function that needs it, leading to potential security issues and difficulties in maintaining the code.
- **Example:** A global variable userBalance might be accessible and modifiable by any function, leading to potential inconsistencies or security risks.

❖ **Linear Code Structure**

- Procedural programs tend to have a linear structure, where the control flow is straightforward and follows the sequence of function calls. This can make the program easier to understand but limits its flexibility.
- **Example:** A program that reads data from a file, processes it, and then writes the output to another file would follow a linear sequence of read(), process(), and write() functions.

These characteristics define how procedural programs are structured and how they operate. While effective for simpler tasks, this approach has limitations when dealing with complex systems, leading many developers to adopt Object-Oriented Programming (OOP) and other modern paradigms.

1.3.2 Problems with the Procedure-Oriented Approach:

The Procedure-Oriented Approach, while effective for smaller and simpler programs, faces several limitations when applied to more complex software development. Here are the key problems associated with the Procedure-Oriented Approach and are described in figure

1. Difficulty in Managing Large Programs

- **Problem:** As the size and complexity of a program increase, it becomes difficult to manage and maintain the code. Functions in procedural programming are often tightly coupled, meaning that changes in one part of the program can have widespread, unintended effects on other parts.
- **Impact:** This tight coupling leads to "spaghetti code," where the program's logic is tangled, making it hard to follow, debug, and maintain.

2. Poor Data Security and Integrity

- **Problem:** Procedural programming typically relies on global variables that can be accessed and modified by any function. This lack of data encapsulation makes it easy for one function to inadvertently alter the data in ways that break the program's logic.
- **Impact:** This can introduce subtle bugs that are hard to track down and fix, especially in large programs where many functions might interact with the same data.

3. Limited Code Reusability

- **Problem:** Functions in procedural programming are often written to handle specific tasks with specific data, making them difficult to reuse in other contexts without significant modification.
- **Impact:** This lack of reusability leads to code duplication, where similar code is written multiple times for slightly different tasks, increasing the program's size and maintenance burden.

4. Lack of Modularity

- **Problem:** Procedural programs often lack clear modularity because the separation of concerns is not enforced as strictly as in Object-Oriented Programming. Functions are not inherently designed to be independent modules, leading to code that is more monolithic and harder to maintain.
- **Impact:** This reduces the ability to isolate and manage different parts of the program independently, making the program harder to understand and modify.

5. Inflexibility

- **Problem:** Procedural programming is less adaptable to changes. Adding new features or modifying existing ones often requires changes across multiple functions and global data structures.
- **Impact:** This can make the program brittle and prone to errors when changes are made, reducing its long-term viability and maintainability.

6. Challenges in Modeling Real-World Problems

- **Problem:** Procedural programming focuses on functions and procedures, which do not naturally correspond to real-world entities and their interactions. This makes it difficult to model complex systems where objects with attributes and behaviors interact in dynamic ways.
- **Impact:** This disconnect between the problem domain and the program structure can lead to less intuitive code, making it harder to understand and maintain, especially for larger projects.

7. Poor Scalability

- **Problem:** As a procedural program grows, the difficulty of managing the codebase increases exponentially. The lack of clear modularity, encapsulation, and reusability makes it hard to scale the program effectively.

- **Impact:** This can limit the program's ability to evolve and grow over time, making it less suitable for long-term projects or systems that require continuous development.

8. Increased Risk of Errors

- **Problem:** Because functions in procedural programming often interact with global data and other functions in complex ways, the risk of introducing errors during development and maintenance is higher.
- **Impact:** This increases the overall cost of development and maintenance, as more time must be spent on debugging and testing to ensure the program behaves as expected.

These problems are some of the key reasons why many developers and organizations have moved towards Object-Oriented Programming (OOP) and other modern paradigms that provide better support for modularity, encapsulation, reusability, and scalability.

1.4 FEATURES OF OBJECT-ORIENTED PROGRAMMING

Here are the key features of Object-Oriented Programming (OOP) listed in a concise, point-wise format and shown in Table 1.1:

Table 1.1 Features of OOP

S.No.	Feature	Description
1.	Encapsulation	Combine data with functions
2.	Abstraction	Hides implementation
3.	Inheritance	Inherits super class properties to sub class
4.	Polymorphism	Object with different forms
5.	Modularity	Split program into parts
6.	Reusability	Reuse of existing code
7.	Dynamic Binding	Runtime Polymorphism
8.	Message Passing	Objects communication

1. **Encapsulation:** Bundles data (attributes) and methods (functions) into a single unit (class) and restricts access to some components to protect data integrity.
2. **Abstraction:** Hides complex implementation details and exposes only the necessary parts, allowing focus on what an object does rather than how it does it.
3. **Inheritance:** Allows a new class to inherit properties and behaviors from an existing class, promoting code reusability and reducing redundancy.
4. **Polymorphism:** Enables objects of different classes to respond to the same function call in different ways, enhancing flexibility and scalability.
5. **Classes and Objects:** Classes serve as blueprints for creating objects, which are instances containing both data and methods that manipulate that data.
6. **Modularity:** Encourages the division of a program into smaller, self-contained modules (classes), improving code organization, readability, and maintainability.
7. **Reusability:** Facilitates the reuse of existing code in new applications, saving time and reducing errors.
8. **Dynamic Binding:** Determines the method to be invoked at runtime, providing flexibility and supporting polymorphism.
9. **Message Passing:** Objects communicate by sending messages (function calls) to each other, enabling complex behaviors through object interactions.

1.5 PROCEDURAL ORIENTED PROGRAMMING VS OBJECT-ORIENTED PROGRAMMING

Table 1.2 Differences between POA and OOP

Procedural -Oriented Programming	Object-Oriented Programming
• Program is split into functions	• Program is split into objects
• Top-down approach	• Bottom-up approach
• No access specifiers	• Has access specifiers
• Less secure	• More Secure
• Overload is not possible	• Overload is possible
• No data hiding	• Has data hiding
• No inheritance	• Has inheritance
• Focus more on function	• Focus more on data
• No code reusability	• Has code reusability

1.6 APPLICATION OF OOP

Here are the key applications of Object-Oriented Programming (OOP) listed in a concise, point-wise format:

- ❖ **Software Development:**
 - OOP is widely used in developing large-scale software systems, including enterprise applications, due to its modularity, reusability, and scalability.
- ❖ **Game Development:**
 - OOP is ideal for creating complex game environments where characters, objects, and interactions are modeled as objects with attributes and behaviors.
- ❖ **Graphical User Interface (GUI) Design:**
 - OOP facilitates the development of GUI applications where elements like buttons, windows, and dialogs are treated as objects that can be manipulated independently.
- ❖ **Simulation and Modeling:**
 - OOP is used in simulations (e.g., flight simulators, scientific models) where real-world entities and their interactions are represented as objects.
- ❖ **Web Development:**
 - OOP principles are used in web development frameworks and languages (like JavaScript, Python, and PHP) to create dynamic, object-oriented web applications.
- ❖ **Database Management Systems (DBMS):**
 - OOP is used in developing DBMS software where data can be modeled as objects, allowing for complex data relationships and operations.
- ❖ **Real-time Systems:**
 - OOP is applied in developing real-time systems, such as operating systems and embedded systems, where modularity and efficient code management are critical.
- ❖ **Distributed Systems:**
 - OOP is used in building distributed systems where objects can communicate across networks, facilitating the development of scalable, distributed applications.
- ❖ **Artificial Intelligence (AI) and Machine Learning (ML):**
 - OOP is used to develop AI and ML models, where different components (e.g., data processing, model training, prediction) are encapsulated as objects.
- ❖ **Mobile Application Development:**
 - OOP is integral to developing mobile apps, where different components of the app are encapsulated into objects, improving code manageability and reusability.

These applications demonstrate the versatility and effectiveness of OOP in various domains, making it a foundational paradigm in modern software development.

1.7 SUMMARY

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects, which are instances of classes. It emphasizes key concepts like encapsulation, abstraction, inheritance, and polymorphism. Encapsulation bundles data and methods together, while abstraction hides complex implementation details. Inheritance allows new classes to inherit properties from existing ones, promoting code reuse. Polymorphism enables objects to be treated as instances of their parent class, allowing for flexible code. OOP is widely used in modern software development due to its modularity, reusability, and ability to model real-world scenarios effectively.

1.8 TECHNICAL TERMS

- Object
- Class
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Method
- Attribute
- Dynamic Binding and etc.

1.9 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain the principles of Object-Oriented Programming with examples.
2. Discuss the advantages of using OOP over procedural programming.
3. Describe the concept of inheritance in OOP with an example.
4. How does polymorphism enhance the flexibility of a program? Provide an example.
5. Explain encapsulation and its importance in software development.

Short questions:

1. What is an object in OOP?
2. Define a class in the context of OOP.
3. What is encapsulation?
4. Explain the concept of inheritance.
5. What is polymorphism in OOP?

1.10 SUGGESTED READINGS

1. "Java: The Complete Reference" by Herbert Schildt, 12th Edition (2021), McGraw-Hill Education
2. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition (2005), O'Reilly Media
3. "Effective Java" by Joshua Bloch, 3rd Edition (2018), Addison-Wesley Professional
4. "Object-Oriented Analysis and Design with Applications" by Grady Booch, 3rd Edition (2007), Addison-Wesley Professional
5. "Thinking in Java" by Bruce Eckel, 4th Edition (2006), Prentice Hall

Dr. KAMPA LAVANYA

LESSON- 2

INTRODUCTION TO JAVA

OBJECTIVES

By the end of this chapter, you should be able to:

- Understand Core Java Concepts.
- Develop Problem-Solving Skills Using Java.
- Build Foundational Programming Skills.
- Encourage Practical Application of Java Knowledge.
- Prepare for Advanced Java and Programming Studies.

STRUCTURE

- 2.1 Introduction
- 2.2 Features of JAVA
- 2.3 Parts of JAVA
 - 2.3.1 Java Development Kit (JDK)
 - 2.3.2 Java Runtime Environment (JRE)
 - 2.3.3 Java Virtual Machine (JVM)
 - 2.3.4 Java Application Programming Interface (API)
- 2.4 Java Naming Conventions
- 2.5 Data Types
- 2.6 Operators
- 2.7 Input and Output Statements
- 2.8 Command Line Arguments
- 2.9 Summary
- 2.10 Technical Terms
- 2.11 Self-Assessment Questions
- 2.12 Suggested Readings

2.1. INTRODUCTION

Java is a versatile, high-level programming language that emphasizes platform independence using the Java Virtual Machine (JVM), which allows Java programs to run on any device with a compatible JVM. The language is structured into several key parts: the Java Development Kit (JDK) for development, the Java Runtime Environment (JRE) for running applications, and the JVM itself for executing compiled bytecode. Java adheres to specific naming conventions to ensure code readability, such as using Pascal Case for class names and camelCase for method and variable names. It supports various data types, including integers, floats, characters, and Booleans, and provides operators for performing arithmetic, relational, and logical operations. Input and output in Java are handled through statements like System.out.println() and Scanner, enabling interaction with users. Additionally, Java allows the use of command-line arguments, enabling users to pass parameters to a program at runtime, enhancing flexibility and control in program execution.

This chapter introduces Java, covers the fundamental features of java, parts of Java, naming conventions, data types, operators, input and output statements, and command line arguments.

2.2 JAVA FEATURES

Java is a powerful and versatile programming language known for its rich set of features that make it one of the most popular choices for developers worldwide. Below are some of the key features that define Java and are shown in Figure 2.1 :

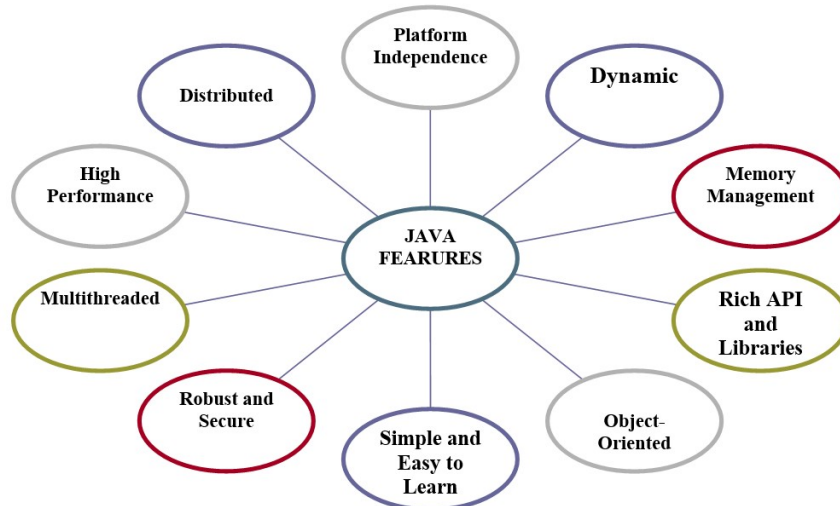


Figure 2.1 Feature of Java

❖ **Platform Independence:**

Java's most celebrated feature is its ability to run on any device with a Java Virtual Machine (JVM). This "write once, run anywhere" (WORA) capability ensures that Java applications are portable across different environments, from desktops to servers to mobile devices.

❖ **Object-Oriented:**

Java is an object-oriented programming language, which means it organizes software design around objects, rather than functions and logic. This approach encourages modular and reusable code, making Java programs easier to maintain and scale.

❖ **Simple and Easy to Learn:**

Java is designed to be easy to use, with a syntax that is clean and easy to understand, especially for those familiar with other programming languages like C or C++. Java removes complex features like pointers and operator overloading, simplifying the learning curve.

❖ **Robust and Secure:**

Java is designed with a strong focus on error handling and runtime checking, making it less prone to crashes and runtime errors. It also includes features like garbage collection to manage memory automatically. Java's security model, including the JVM's ability to sandbox applications, makes it a secure choice for developing applications, especially for web-based environments.

❖ **Multithreaded:**

Java natively supports multithreading, allowing the concurrent execution of two or more threads. This feature is crucial for developing applications that need to perform multiple tasks simultaneously, such as games, server applications, and real-time systems.

❖ **High Performance:**

While Java is an interpreted language, the introduction of Just-In-Time (JIT) compilers and performance enhancements in the JVM allows Java applications to run with high efficiency, making it competitive with natively compiled languages.

❖ **Distributed:**

Java is designed for distributed computing, enabling developers to create applications that can run across networks and interact with other services. It provides robust support for

networking through the java.net package and APIs for Remote Method Invocation (RMI) and Enterprise JavaBeans (EJB).

❖ **Dynamic:**

Java is a dynamic language, capable of adapting to an evolving environment. It supports dynamic loading of classes and functions, allowing for the development of flexible and extensible programs. Java programs can also adapt to new environments and systems without requiring changes in the source code.

❖ **Memory Management:**

Java provides automated memory management through its garbage collection mechanism, which automatically removes objects that are no longer in use. This reduces the burden on developers to manage memory manually, minimizing memory leaks and other memory-related issues.

❖ **Rich API and Libraries:**

Java comes with a vast set of APIs and libraries that provide ready-to-use functions for various tasks, from data structures and algorithms to networking and database management. This extensive library support accelerates development and reduces the need for third-party libraries.

These features collectively make Java a preferred language for a wide range of applications, from web and mobile applications to enterprise-level systems and scientific computing.

Table 2.1 Features of Java

S.No	Feature	Description
1.	Simple	Java is easy to learn, and its syntax is quite simple, clean and easy to understand
2.	Object Oriented	Java can be easily extended as it is based on Object Model
3.	Robust	automatic Garbage Collector and Exception Handling.
4.	Platform Independent	bytecode is platform independent and can be run on any machine, allow security
5.	Multi-Threading	utilizes same memory and other resources to execute multiple threads at the same time
6.	High Performance	the use of just-in-time compiler
7.	Distributed	designed to run on computer networks

2.3 PARTS OF JAVA

Java is composed of several key parts that work together to enable the development, execution, and management of Java applications. Here’s an overview of the main parts of Java:

2.3.1 Java Development Kit (JDK):

The Java Development Kit (JDK) is a comprehensive suite of tools and libraries necessary for developing Java applications. It is the primary component for Java developers, offering everything required to write, compile, debug, and run Java applications. The JDK is available for various operating systems, including Windows, macOS, and Linux, ensuring that Java development can be done across multiple platforms.

Key Components of the JDK:

❖ **Compiler (javac):**

- The Java compiler (javac) converts Java source code (.java files) into bytecode (.class files). This bytecode is platform-independent and can be executed by the Java Virtual Machine (JVM). The compiler checks the code for errors and ensures that it adheres to Java syntax and rules before generating bytecode.

❖ **Java Runtime Environment (JRE):**

- The JRE is bundled with the JDK and provides the runtime environment necessary to execute Java applications. It includes the JVM, core libraries, and other supporting components that allow Java programs to run. While the JRE can run Java applications, it does not include tools for developing them.

❖ **Debugger (jdb):**

- The Java debugger (jdb) is a tool used to find and fix bugs in Java programs. It allows developers to step through code, set breakpoints, inspect variables, and evaluate expressions, making it easier to identify and resolve issues in the code.

❖ **JavaDoc:**

- JavaDoc is a documentation generator that creates API documentation in HTML format from comments in the source code. This tool is essential for generating clear and professional documentation, which is crucial for maintaining and sharing code with others.

❖ **Java Archive Tool (jar):**

- The Java Archive tool (jar) is used to package Java classes and associated resources (such as images and text files) into a single archive file with a .jar extension. This archive can be used to distribute and deploy Java applications or libraries. The JAR format also supports compression, reducing the file size of the archive.

❖ **Java Virtual Machine (JVM):**

- While the JVM is technically part of the JRE, it is included in the JDK as well, since the JDK encompasses all the components required to both develop and run Java applications. The JVM is responsible for executing the bytecode produced by the Java compiler.

❖ **Additional Tools:**

- The JDK includes several other tools that assist in Java development, such as:
 - javap: A class file disassembler that helps developers understand the structure of compiled classes.
 - jconsole: A monitoring tool that provides information about the performance and resource consumption of Java applications.
 - jarsigner: A tool for signing JAR files, ensuring the authenticity and integrity of the code within them.

The JDK is essential for Java developers, as it provides all the tools necessary for writing, compiling, debugging, and running Java applications. Whether developing simple applications or complex enterprise-level systems, the JDK offers the flexibility and functionality required to manage the entire software development lifecycle in Java.

2.3.2 Java Runtime Environment (JRE):

The Java Runtime Environment (JRE) is a crucial component of the Java platform, providing the necessary environment for running Java applications. It includes everything required to execute Java programs but does not contain the development tools (like the compiler) found in the Java Development Kit (JDK). The JRE is used by end-users who need to run Java programs but do not need to develop them.

Key Components of the JRE:**❖ Java Virtual Machine (JVM):**

- The JVM is the core of the JRE and is responsible for executing Java bytecode. It provides the abstraction that allows Java programs to run on any platform, making Java platform independent. The JVM interprets or compiles bytecode into machine code that the operating system can execute. It also manages memory allocation, garbage collection, and runtime security checks.

❖ Core Libraries:

- The JRE includes a comprehensive set of libraries that provide the necessary functionality to run Java applications. These libraries include essential classes and APIs that Java programs use to perform tasks such as:
 - Input/Output Operations: Libraries like `java.io` for reading from and writing to files.
 - Networking: Libraries like `java.net` for networking capabilities.
 - Utilities: Libraries like `java.util` that provide data structures (e.g., collections), date and time utilities, and other common utilities.
 - Math Functions: Libraries like `java.math` for complex mathematical operations.

❖ Java Class Loader:

- The class loader is part of the JVM that dynamically loads Java classes into memory as needed. It allows Java applications to load classes from various sources, such as the local file system, network, or even a remote server. The class loader also ensures that the correct version of a class is loaded, which is essential for maintaining application stability.

❖ Java Security Manager:

- The security manager in the JRE controls access to system resources by Java applications. It enforces security policies that prevent potentially harmful operations, such as reading or writing to files, accessing network connections, or executing certain native code. This is particularly important for running Java applets or applications from untrusted sources.

❖ Java Plug-in:

- The Java plug-in enables web browsers to run Java applets. It allows Java applications embedded in web pages to be executed within the browser environment. Although less commonly used today with the decline of applets, it was an essential part of Java's role in web development.

❖ Java Web Start:

- Java Web Start allows users to run Java applications directly from the web. It downloads the necessary Java application files from the internet and launches them, simplifying the distribution and installation of Java applications.

The JRE is essential for running Java applications on any device. It is used by end-users who need to execute Java programs but do not need to write or compile Java code. The JRE ensures that Java applications can run consistently across different platforms by providing a standardized runtime environment. Whether it's a desktop application, a server-side application, or a small applet, the JRE provides the necessary components to run the Java code reliably and securely. In summary, the JRE is a runtime environment that includes the JVM, core libraries, and other components needed to execute Java applications, ensuring that Java programs can run on any platform with a compatible JRE installed.

2.3.3 Java Virtual Machine (JVM):

The Java Virtual Machine (JVM) is a critical component of the Java platform, responsible for executing Java bytecode on any device or operating system. The complete architecture of JVM is shown in Figure 2.2. The JVM provides an abstraction layer that allows Java applications to be "write once, run anywhere," meaning that the same Java program can run on any platform without modification, if a compatible JVM is available.

Key Components and Functions of the JVM:

❖ **Class Loader:**

- The class loader is a part of the JVM that loads Java class files into memory. It reads the .class files containing Java bytecode and brings them into the runtime environment. The class loader also performs tasks such as verifying the bytecode and linking the classes by resolving references to other classes, ensuring that all dependencies are satisfied before execution begins.

❖ **Runtime Memory Areas (JVM Memory):**

- The JVM manages memory during the execution of Java programs using different memory areas:
 - **Heap:** The heap is where all the objects and their corresponding instance variables are stored. It is shared among all threads.
 - **Stack:** Each thread in a Java application has its own stack. The stack stores method call frames, including local variables and partial results. The stack is also where method return values are stored.
 - **Method Area:** The method area is a shared memory space that stores class-level data such as the runtime constant pool, field and method data, and code for methods.
 - **Program Counter (PC) Register:** The PC register keeps track of the current instruction being executed in the thread.
 - **Native Method Stack:** This stack is used for native method calls, which are methods written in languages other than Java, such as C or C++.

❖ **Execution Engine:**

- The execution engine is the core of the JVM, responsible for executing the bytecode instructions. It consists of several components:
 - **Interpreter:** The interpreter reads and executes the bytecode instructions one at a time. While the interpreter is simple and fast, it can be slower because it processes each instruction individually.
 - **Just-In-Time (JIT) Compiler:** To improve performance, the JIT compiler compiles frequently executed bytecode sequences into native machine code at runtime. This compiled code is then executed directly by the CPU, leading to significant performance gains.
 - **Garbage Collector:** The garbage collector automatically manages memory by identifying and disposing of objects that are no longer in use, freeing up memory and preventing memory leaks.

❖ **Java Native Interface (JNI):**

- The JNI is an interface that allows Java code running in the JVM to call and be called by native applications and libraries written in other languages like C, C++, or assembly. This capability is important for integrating Java applications with legacy systems or specialized hardware.

❖ **Native Method Libraries:**

- These are libraries that contain native code, usually written in languages other than Java, which the JVM can call. These libraries are platform-specific and provide low-level operations that are not possible in standard Java.

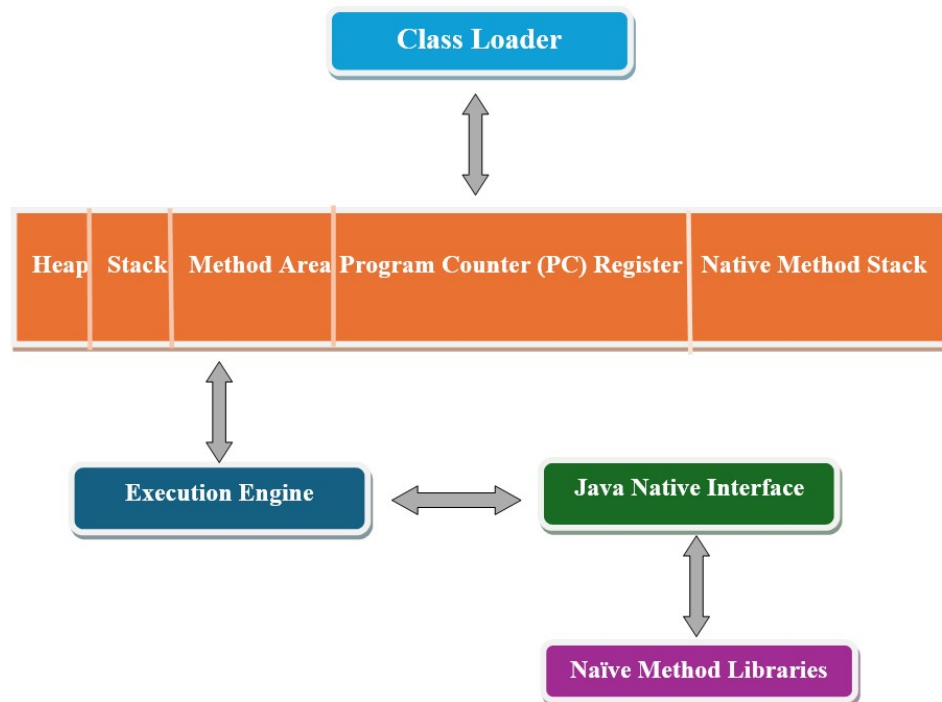


Fig 2.2 The Architecture of JVM with Key Components

Key Characteristics of the JVM:

❖ **Platform Independence:**

- The JVM is what makes Java platform independent. By translating Java bytecode into machine-specific code, the JVM allows the same Java program to run on different types of hardware and operating systems without modification.

❖ **Memory Management:**

- The JVM manages memory through its garbage collection mechanism, which automatically reclaims memory by removing objects that are no longer in use. This helps prevent memory leaks and improves application stability.

❖ **Security:**

- The JVM includes built-in security features, such as the bytecode verifier, which checks for illegal code that could violate access rights or cause security issues. The security manager and class loaders further enhance the security by controlling access to system resources and ensuring that only trusted code is executed.

The JVM is the cornerstone of the Java platform, providing the environment in which Java programs execute. Its ability to abstract the underlying hardware and operating system allows Java developers to focus on writing code without worrying about platform-specific details. The JVM’s features, such as garbage collection, JIT compilation, and security mechanisms, contribute to Java’s reputation for being robust, secure, and high-performance.

The JVM is a powerful and flexible component that enables the execution of Java applications across various platforms, managing memory, ensuring security, and optimizing performance through advanced execution techniques.

2.3.4 Java Application Programming Interface (API):

The Java Application Programming Interface (API) is a vast collection of pre-written packages, classes, and interfaces that provide developers with ready-to-use functionalities for various tasks. The Java API simplifies the process of writing Java applications by offering reusable code components, enabling developers to focus on the unique aspects of their applications rather than reinventing the wheel for common tasks.

Key Components of the Java API:

❖ Core Libraries:

- The core libraries form the foundation of the Java API, providing essential classes and interfaces needed for basic programming tasks. These libraries include:
 - `java.lang`: Contains fundamental classes such as `String`, `Math`, `Integer`, and `Thread`, which are automatically imported into every Java program.
 - `java.util`: Provides utility classes for data structures (like `ArrayList`, `HashMap`), date and time manipulation, random number generation, and more.
 - `java.io`: Offers classes for input and output operations, including reading from and writing to files, handling streams, and performing serialization.

❖ Networking Libraries:

- Java's networking libraries, found primarily in the `java.net` package, enable developers to build networked applications. These libraries support operations like connecting to remote servers, sending and receiving data over the network, handling URLs, and implementing sockets for communication.

❖ Database Connectivity (JDBC):

- The Java Database Connectivity (JDBC) API, located in the `java.sql` package, allows Java applications to interact with relational databases. JDBC provides methods to connect to a database, execute SQL queries, and retrieve and manipulate data from a database.

❖ User Interface (UI) Libraries:

- Java offers several APIs for building graphical user interfaces (GUIs):
 - `AWT` (Abstract Window Toolkit): Provides the basic components for building simple graphical user interfaces.
 - `Swing`: An extension of `AWT`, `Swing` provides a richer set of components for building more sophisticated UIs, such as buttons, tables, text fields, and more.
 - `JavaFX`: A more modern library for creating rich internet applications (RIAs) with advanced UI features, including 2D/3D graphics, media playback, and web rendering.

❖ Concurrency and Multithreading:

- The `java.util.concurrent` package contains classes and interfaces that simplify the development of multithreaded applications. It includes utilities for managing threads, synchronizing data between threads, and implementing high-level concurrency patterns like executors, thread pools, and concurrent data structures.

❖ Security Libraries:

- Java's security API, primarily found in the `java.security` package, offers tools for implementing security features in Java applications. This includes classes for encryption, decryption, key generation, digital signatures, and secure random number generation. Java also provides APIs for managing authentication and authorization through the Java Authentication and Authorization Service (JAAS).

❖ XML Processing:

- Java provides APIs for working with XML, such as the `javax.xml.parsers` package for parsing XML documents, the `javax.xml.bind` (JAXB) package for binding XML documents to Java objects, and the `javax.xml.transform` package for transforming XML documents.

❖ Web and Enterprise Development:

- Java has a robust set of APIs for developing web and enterprise applications:
 - Servlets and JSP: Found in the `javax.servlet` and `javax.servlet.jsp` packages, these APIs allow developers to create dynamic web content using Java.
 - Java EE (Enterprise Edition): Provides a comprehensive set of APIs for building large-scale, distributed, and transactional applications, including technologies like EJB (Enterprise JavaBeans), JMS (Java Message Service), and JPA (Java Persistence API).

❖ Remote Method Invocation (RMI):

- The RMI API, found in the `java.rmi` package, allows Java applications to invoke methods on remote objects, enabling distributed computing. This API is used to create applications that can communicate over a network, with objects residing on different machines.

The Java API is a critical component of the Java platform, enabling developers to write powerful applications without needing to implement every functionality from scratch. By providing a rich set of pre-built classes and interfaces, the Java API accelerates development, enhances code reliability, and promotes the reuse of well-tested code components. Whether working on simple applications or complex enterprise systems, developers can rely on the Java API to provide the tools and resources needed to build robust and efficient software. The Java API is an extensive and versatile toolkit that provides everything from basic programming constructs to advanced functionalities for networking, database access, user interface design, and more, making it an essential resource for Java developers.

2.4 NAMING CONVENTIONS

Java naming conventions are guidelines for naming various elements in a Java program, ensuring that code is consistent, readable, and maintainable. Adhering to these conventions is crucial, especially when working in teams or contributing to large codebases.

1. Class Names:

- Convention: Use PascalCase (also known as UpperCamelCase).
- Description: Each word in the class name starts with an uppercase letter. Class names should typically be nouns or noun phrases, as they represent objects or entities.
- Example: Customer, EmployeeDetails, InvoiceProcessor.

2. Method Names:

- Convention: Use camelCase.
- Description: Start with a lowercase letter, and capitalize the first letter of each subsequent word. Method names should typically be verbs or verb phrases, as they represent actions or behaviors.
- Example: calculateTotal, getEmployeeName, processInvoice.

3. Variable Names:

- Convention: Use camelCase.

- Description: Like method names, variable names start with a lowercase letter, and subsequent words are capitalized. Variable names should be descriptive and represent the data they hold.
- Example: totalAmount, employeeName, invoiceList.

4. Constant Names:

- Convention: Use ALL_UPPERCASE with words separated by underscores.
- Description: Constants are usually declared using the final keyword and should be named in all uppercase letters to distinguish them from variables. Each word is separated by an underscore.
- Example: MAX_HEIGHT, DEFAULT_TIMEOUT, PI.

5. Package Names:

- Convention: Use lowercase letters.
- Description: Package names are typically written in all lowercase letters and often follow the reverse domain name convention to avoid name conflicts. Words in package names are usually separated by periods.
- Example: com.example.projectname, org.companyname.module.

6. Interface Names:

- Convention: Use PascalCase.
- Description: Interface names should be written like class names, using PascalCase. They often represent capabilities or behaviors, and in some cases, they may be adjectives.
- Example: Runnable, Serializable, Comparable.

7. Enum Names:

- Convention: Use PascalCase for the enum name and ALL_UPPERCASE for the enum constants.
- Description: The name of the enum itself follows the same convention as classes, while the constants within the enum are typically named using uppercase letters with underscores separating words.
- Example: enum DayOfWeek { SUNDAY, MONDAY, TUESDAY }, enum Color { RED, GREEN, BLUE }.

8. Type Parameter Names:

- Convention: Use single uppercase letters.
- Description: In generic types or methods, type parameters are usually named with single, uppercase letters. Commonly used letters include T for type, E for element, K for key, and V for value.
- Example: class Box<T>, interface List<E>, Map<K, V>.

Importance of Following Naming Conventions:

- Consistency: Uniform naming across a codebase helps in understanding and maintaining the code.
- Readability: Clear and descriptive names make it easier for developers to understand what a piece of code does.
- Maintainability: Well-named classes, methods, and variables reduce the cognitive load on developers, making it easier to navigate and modify the code over time.
- Collaboration: When working in teams, consistent naming conventions ensure that everyone can read and understand each other's code, reducing the chances of errors and misunderstandings.

2.5 DATA TYPES

In Java, data types specify the size and type of values that can be stored in variables. Java is a statically typed language, meaning that each variable must be declared with a data type before it can be used. Java's data types are categorized into two main groups: primitive data types and non-primitive data types and are shown in Figure 2.3.

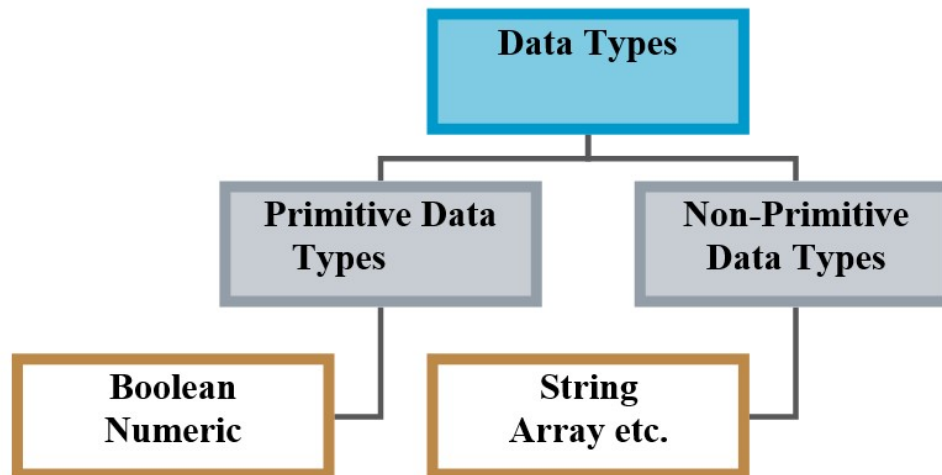


Figure 2.3 Classification of Java Data Types

❖ Primitive Data Types:

- Primitive data types are the most basic data types available in Java. They are predefined by the language and named by a keyword. Java has eight primitive data types which are shown in Table 2.2:
 - **byte:**
 - **Size:** 8 bits
 - **Range:** -128 to 127
 - **Description:** Useful for saving memory in large arrays, where the memory savings are most needed. It can also be used in place of int where the range of values is known to be small.
 - **Example:** byte b = 100;
 - **short:**
 - **Size:** 16 bits
 - **Range:** -32,768 to 32,767
 - **Description:** A data type that is larger than byte but smaller than int. It's also used to save memory in large arrays.
 - **Example:** short s = 10000;
 - **int:**
 - **Size:** 32 bits
 - **Range:** -2^{31} to $2^{31}-1$ (-2,147,483,648 to 2,147,483,647)
 - **Description:** The default choice for integral values unless there is a reason to use byte or short. Most commonly used for integer arithmetic.
 - **Example:** int i = 100000;
 - **long:**
 - **Size:** 64 bits
 - **Range:** -2^{63} to $2^{63}-1$ (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
 - **Description:** Used when a wider range than int is needed.

- **Example:** long l = 100000L;
- **float:**
- **Size:** 32 bits
- **Range:** Varies, approximately $\pm 3.40282347E+38F$ (6-7 significant decimal digits)
- **Description:** Used for single-precision floating-point numbers. It's recommended to use float if you need to save memory in large arrays of floating-point numbers.
- **Example:** float f = 234.5f;
- **double:**
- **Size:** 64 bits
- **Range:** Varies, approximately $\pm 1.79769313486231570E+308$ (15 significant decimal digits)
- **Description:** Used for double-precision floating-point numbers and is the default choice for decimal values.
- **Example:** double d = 123.456;
- **char:**
- **Size:** 16 bits (2 bytes)
- **Range:** 0 to 65,535 (unsigned)
- **Description:** Used to store a single character. Java uses Unicode, so it can store any character from any language.
- **Example:** char c = 'A';
- **boolean:**
- **Size:** Not precisely defined (depends on JVM implementation, but typically 1 bit)
- **Range:** true or false
- **Description:** Used for flags that track true/false conditions.
- **Example:** boolean isJavaFun = true;

Table 2.2 Primitive Data Types in Java

Data Type	Default Value	Default size	Range
byte	0	1 byte or 8 bits	-128 to 127
short	0	2 bytes or 16 bits	-32,768 to 32,767
int	0	4 bytes or 32 bits	2,147,483,648 to 2,147,483,647
long	0	8 bytes or 64 bits	9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	0.0f	4 bytes or 32 bits	1.4e-045 to 3.4e+038
double	0.0d	8 bytes or 64 bits	4.9e-324 to 1.8e+308
char	'\u0000'	2 bytes or 16 bits	0 to 65536
boolean	FALSE	1 byte or 2 bytes	0 or 1

❖ Non-Primitive Data Types

- Non-Primitive data types are not predefined like primitive data types. Instead, they are created by the programmer and can refer to any object in Java. Reference variables store the memory address of the object they refer to, rather than the data itself.
- **Strings:**
- **Description:** Strings are objects in Java, represented by the String class. They are used to store sequences of characters.
- **Example:** String message = "Hello, World!";
- **Arrays:**
- **Description:** Arrays are objects that store multiple variables of the same type. The size of an array is fixed upon creation.

- **Example:** `int[] numbers = {1, 2, 3, 4, 5};`
- **Classes and Objects:**
- **Description:** Classes define new data types by grouping data and methods that operate on the data. When you create an instance of a class, it is called an object.

- **Example:**

java

Copy code

```
class Car {
    String model;
    int year;
}
Car myCar = new Car();
myCar.model = "Tesla";
myCar.year = 2021;
```

- **Interfaces:**

- **Description:** Interfaces define a contract or a set of methods that a class must implement. They are used to achieve abstraction and multiple inheritance in Java.

- **Example:**

java

Copy code

```
interface Vehicle {
    void start();
}
class Bike implements Vehicle {
    public void start() {
        System.out.println("Bike started");
    }
}
```

Understanding and properly using data types is fundamental in Java programming. The correct data type ensures that you use memory efficiently and avoid errors. Primitive types are straightforward and efficient for basic data handling, while non-primitive types allow for more complex data structures and operations.

2.6 OPERATORS

Operators in Java are special symbols or keywords used to perform operations on variables and values. Java provides a rich set of operators to manipulate data and variables, ranging from simple arithmetic to complex logical operations. These operators are grouped into several categories based on their functionality.

1. Arithmetic Operators:

- **Description:** These operators perform basic arithmetic operations such as addition, subtraction, multiplication, and division.
- **Operators and Examples:**
 - **+** **(Addition):** Adds two operands.
Example: `int sum = 5 + 3; // sum = 8`
 - **- (Subtraction):** Subtracts the right operand from the left operand.
Example: `int difference = 5 - 3; // difference = 2`

- ***** **(Multiplication):** Multiplies two operands.
Example: `int product = 5 * 3; // product = 15`
- **/** **(Division):** Divides the left operand by the right operand.
Example: `int quotient = 6 / 3; // quotient = 2`
- **%** **(Modulus):** Returns the remainder of a division.
Example: `int remainder = 7 % 3; // remainder = 1`

2. Assignment Operators:

- **Description:** These operators are used to assign values to variables.
- **Operators and Examples:**
 - **= (Assignment):** Assigns the value on the right to the variable on the left.
Example: `int a = 5;`
 - **+= (Add and Assign):** Adds the right operand to the left operand and assigns the result to the left operand.
Example: `a += 3; // a = a + 3, so a becomes 8`
 - **-= (Subtract and Assign):** Subtracts the right operand from the left operand and assigns the result to the left operand.
Example: `a -= 2; // a = a - 2, so a becomes 6`
 - ***= (Multiply and Assign):** Multiplies the left operand by the right operand and assigns the result to the left operand.
Example: `a *= 2; // a = a * 2, so a becomes 12`
 - **/= (Divide and Assign):** Divides the left operand by the right operand and assigns the result to the left operand.
Example: `a /= 3; // a = a / 3, so a becomes 4`
 - **%= (Modulus and Assign):** Takes the modulus of the left operand by the right operand and assigns the result to the left operand.
Example: `a %= 3; // a = a % 3, so a becomes 1`

3. Relational Operators:

- **Description:** These operators compare two values and return a boolean result (true or false).
- **Operators and Examples:**
 - **== (Equal to):** Checks if two values are equal.
Example: `boolean isEqual = (5 == 3); // isEqual = false`
 - **!= (Not Equal to):** Checks if two values are not equal.
Example: `boolean isNotEqual = (5 != 3); // isNotEqual = true`
 - **> (Greater than):** Checks if the left operand is greater than the right operand.
Example: `boolean isGreater = (5 > 3); // isGreater = true`
 - **< (Less than):** Checks if the left operand is less than the right operand.
Example: `boolean isLess = (5 < 3); // isLess = false`
 - **>= (Greater than or Equal to):** Checks if the left operand is greater than or equal to the right operand.
Example: `boolean isGreaterOrEqual = (5 >= 3); // isGreaterOrEqual = true`
 - **<= (Less than or Equal to):** Checks if the left operand is less than or equal to the right operand.
Example: `boolean isLessOrEqual = (5 <= 3); // isLessOrEqual = false`

4. Logical Operators:

- **Description:** These operators are used to perform logical operations on boolean values.

- **Operators and Examples:**

- **&& (Logical AND):** Returns true if both operands are true.
Example: `boolean result = (5 > 3 && 8 > 6); // result = true`
- **|| (Logical OR):** Returns true if at least one of the operands is true.
Example: `boolean result = (5 > 3 || 8 < 6); // result = true`
- **! (Logical NOT):** Reverses the logical state of its operand.
Example: `boolean result = !(5 > 3); // result = false`

5. Unary Operators:

- **Description:** These operators operate on a single operand.
- **Operators and Examples:**
 - **+ (Unary Plus):** Indicates a positive value (typically optional as numbers are positive by default).
Example: `int positive = +5;`
 - **- (Unary Minus):** Negates the value of the operand.
Example: `int negative = -5;`
 - **++ (Increment):** Increases the value of the operand by 1.
Example: `int a = 5; a++; // a becomes 6`
 - **-- (Decrement):** Decreases the value of the operand by 1.
Example: `int a = 5; a--; // a becomes 4`
 - **! (Logical NOT):** Inverts the value of a boolean operand.
Example: `boolean isTrue = true; isTrue = !isTrue; // isTrue becomes false`

6. Bitwise Operators:

- **Description:** These operators perform bit-level operations on integer types.
- **Operators and Examples:**
 - **& (Bitwise AND):** Performs a bitwise AND operation on two operands.
Example: `int result = 5 & 3; // result = 1 (0101 & 0011 = 0001)`
 - **| (Bitwise OR):** Performs a bitwise OR operation on two operands.
Example: `int result = 5 | 3; // result = 7 (0101 | 0011 = 0111)`
 - **^ (Bitwise XOR):** Performs a bitwise XOR operation on two operands.
Example: `int result = 5 ^ 3; // result = 6 (0101 ^ 0011 = 0110)`
 - **~ (Bitwise Complement):** Inverts all the bits of the operand.
Example: `int result = ~5; // result = -6 (bitwise complement of 0101)`
 - **<< (Left Shift):** Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
Example: `int result = 5 << 2; // result = 20 (0101 << 2 = 10100)`
 - **>> (Right Shift):** Shifts the bits of the left operand to the right by the number of positions specified by the right operand.
Example: `int result = 5 >> 2; // result = 1 (0101 >> 2 = 0001)`
 - **>>> (Unsigned Right Shift):** Shifts the bits of the left operand to the right by the number of positions specified by the right operand, filling the leftmost bits with zeros.
Example: `int result = 5 >>> 2; // result = 1`

7. Ternary Operator:

- **Description:** The ternary operator is a shorthand for an if-else statement. It has three operands and is used to evaluate a boolean expression.
- **Operator and Example:**

- **? : (Ternary):** Evaluates a condition and returns one of two values depending on whether the condition is true or false.
Example: `int result = (5 > 3) ? 10 : 20; // result = 10`

8. Instanceof Operator:

- **Description:** The instanceof operator checks whether an object is an instance of a specific class or subclass.

- **Operator and Example:**

- **instanceof:** Returns true if the object is an instance of the specified class or subclass, otherwise false.

Example: `boolean isString = "Hello" instanceof String; // isString = true`

Operators are fundamental to performing operations on variables and data in Java. They allow developers to build complex expressions and perform calculations, comparisons, and logical operations. Understanding and using operators correctly is essential for writing effective and efficient Java code.

2.7 INPUT AND OUTPUT STATEMENTS

Java provides various ways to handle input and output (I/O) operations, enabling interaction between the user and the program. Two of the most commonly used tools for this purpose are the Scanner class for input and the System.out.println method for output.

❖ Scanner

The Scanner class in Java is used to read input from various sources, most commonly from the keyboard (standard input). It is a part of the java.util package, so you need to import it before using it.

- **Importing the Scanner Class:**

```
import java.util.Scanner;
```

- **Creating a Scanner Object:** To read input from the keyboard, you need to create a Scanner object that uses System.in as the input stream.

```
Scanner scanner = new Scanner(System.in);
```

- **Reading Different Types of Input:**

- **Reading a String:**

```
System.out.print("Enter your name: ");
```

```
String name = scanner.nextLine(); // Reads a line of text
```

```
System.out.println("Hello, " + name + "!");
```

- **Reading an Integer:**

```
System.out.print("Enter your age: ");
```

```
int age = scanner.nextInt(); // Reads an integer
```

```
System.out.println("You are " + age + " years old.");
```

- **Reading a Double:**

```
System.out.print("Enter your salary: ");
```

```
double salary = scanner.nextDouble(); // Reads a double value
```

```
System.out.println("Your salary is " + salary);
```

- **Reading a Single Word:**

```
System.out.print("Enter your first name: ");
```

```
String firstName = scanner.next(); // Reads a single word (until a space is encountered)
```

```
System.out.println("First Name: " + firstName);
```

- **Key Points:**

- The nextLine() method reads an entire line of input, including spaces.

- The `nextInt()`, `nextDouble()`, etc., methods read specific types of input and automatically convert them to the appropriate data type.
- The `next()` method reads input until a space or newline is encountered, making it useful for single-word inputs.

Output: `System.out.println`

The `System.out.println` method is used to print information to the console. It's one of the most frequently used methods in Java for outputting text, variables, and results of operations.

- **Syntax:**

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

Where `expression` can be a string, a variable, or a combination of strings and variables.

- **Examples:**

```
System.out.println("Hello, World!"); // Prints: Hello, World!
```

```
System.out.println(100);           // Prints: 100
```

```
System.out.println("Total: " + 50); // Prints: Total: 50
```

- **Key Points:**

- The `println` method prints the specified message and then moves the cursor to the next line.
- There is also a `System.out.print` method that prints the message without moving to the next line, allowing multiple outputs to be printed on the same line.

2.8 COMMAND LINE ARGUMENTS

Command line arguments in Java are the inputs that are passed to a Java program during its execution from the command line or terminal. These arguments are typically passed as strings, and they are accessible within the `main` method of the Java program.

Here's a basic overview of how command line arguments work in Java:

- ❖ **Accessing Command Line Arguments**

Command line arguments in Java are stored in the `String` array that is passed to the `main` method of the class. The `main` method signature looks like this:

```
public static void main(String[] args) {  
    // Code goes here  
}
```

Where `args` is an array of `String` objects that contains the arguments passed from the command line.

- ❖ **Example: Simple Java Program with Command Line Arguments**

```
public class CommandLineExample {  
    public static void main(String[] args) {  
        // Check if arguments are passed  
        if (args.length > 0) {  
            System.out.println("Command line arguments are:");  
            for (int i = 0; i < args.length; i++) {  
                System.out.println("Argument " + i + ": " + args[i]);  
            }  
        } else {  
            System.out.println("No command line arguments found.");  
        }  
    }  
}
```

❖ **Running the Program**

To run this program and pass command line arguments, you would use the following command in the terminal or command prompt:

```
java CommandLineExample arg1 arg2 arg3
```

Where arg1, arg2, and arg3 are the command line arguments being passed to the program. The program will output:

Command line arguments are:

Argument 0: arg1

Argument 1: arg2

Argument 2: arg3

Important Points

- **Indexing:** The command line arguments are indexed starting from 0. So args[0] is the first argument, args[1] is the second, and so on.
- **Argument Count:** You can determine the number of arguments by checking args.length.
- **Data Types:** Command line arguments are always passed as strings. If you need them in another data type (e.g., int), you need to parse the string to that type using methods like Integer.parseInt(args[0]).

2.9 SUMMARY

Java is a widely used, object-oriented programming language designed for portability, security, and high performance. It operates on the Java Virtual Machine (JVM), which allows Java programs to run on any platform without modification, making it platform independent.

Java features a rich set of data types, including primitive types like int, float, char, and boolean, as well as reference types like arrays and objects. The language offers a variety of operators, such as arithmetic, relational, and logical operators, to perform operations on variables and data. Command line arguments in Java allow users to pass input to programs via the terminal, accessible through the String[] args parameter in the main method.

Additionally, the Scanner class provides a way to take user input during runtime, while the System.out.println method is commonly used for outputting data to the console, making it easy to display results or messages. Java's combination of simplicity, portability, and powerful features has made it a popular choice for developing everything from mobile applications to enterprise-level systems.

2.10 TECHNICAL TERMS

- JVM
- Data Type
- Scanner
- Println
- Boolean
- Platform Independence
- Portable
- Command Line Argument

2.11 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain the structure and significance of data types in Java programming.
2. Describe the architecture and functionality of the Java Virtual Machine (JVM).
3. Analyze the key features of Java that make it a robust and versatile programming language
4. Discuss the different types of operators in Java and their role in program development.
5. Compare and contrast command line arguments and the Scanner class for input handling in Java.

Short questions:

1. List and briefly describe three key features of Java.
2. What is the purpose of arithmetic operators in Java?
3. How does the Java Virtual Machine (JVM) contribute to Java's platform independence?
4. What is the role of data types in Java?

2.12 SUGGESTED READINGS

1. "Java: The Complete Reference" by Herbert Schildt, 12th Edition (2021), McGraw-Hill Education
2. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition (2005), O'Reilly Media
3. "Effective Java" by Joshua Bloch, 3rd Edition (2018), Addison-Wesley Professional
4. "Object-Oriented Analysis and Design with Applications" by Grady Booch, 3rd Edition (2007), Addison-Wesley Professional
5. "Thinking in Java" by Bruce Eckel, 4th Edition (2006), Prentice Hall

Dr. KAMPA LAVANYA

LESSON- 3

CONDITIONAL STATEMENTS

OBJECTIVES

By the end of this chapter, you should be able to:

- make decisions within a program based on different conditions.
- control the flow of execution by branching the program into different paths.
- program to adapt its behavior based on real-time inputs or data.
- code readability and maintainability by clearly defining different execution paths.
- complex logical operations by combining multiple conditions using logical operators.

STRUCTURE

- 3.1 Introduction
- 3.2 Introduction to Conditional Statements
- 3.3 Basic Conditional Statements
- 3.4 Advanced Conditional Statements
- 3.5 Control Flow Enhancements
- 3.6 Common Use Cases
- 3.7 Best Practices
- 3.8 Summary
- 3.9 Technical Terms
- 3.10 Self-Assessment Questions
- 3.11 Suggested Readings

3.1. INTRODUCTION

Conditional statements in Java, such as if, else if, else, and switch, are used to control the flow of a program based on different conditions. These statements allow a program to execute specific blocks of code only when certain conditions are met, enabling it to make decisions and respond dynamically to various inputs or states. This is crucial for implementing logic that depends on runtime conditions, such as user inputs, data values, or computational results. By using conditional statements, developers can create more flexible and interactive applications that adapt their behavior based on real-time conditions. This chapter introduces Java, covers the fundamental features of java, parts of Java, naming conventions, data types, operators, input and output statements, and command line arguments.

3.2 INTRODUCTION TO CONDITIONAL STATEMENTS

Conditional statements are fundamental constructs in programming languages, including Java, that allow a program to make decisions based on certain conditions. They enable a program to execute specific blocks of code only when certain criteria are met, thus providing a way to branch the flow of execution.

In Java, conditional statements include if, else if, else, switch, and ternary operators. These statements evaluate expressions that return boolean values (true or false) to determine which code blocks to execute. The basic purpose of conditional statements is to enable decision-making in code, allowing the program to respond to different inputs or states.

Importance in Programming

1. **Decision Making:** Conditional statements enable a program to make decisions and choose different execution paths based on various conditions. This decision-making capability is essential for implementing logic that reacts to user inputs, data values, or other dynamic factors.
2. **Control Flow Management:** By controlling the flow of execution, conditional statements help manage the sequence of operations in a program. This allows developers to implement complex logic and ensure that the program performs the correct actions under different scenarios.
3. **Dynamic Behavior:** Conditional statements allow a program to adapt its behavior in real-time. For example, they can enable different features based on user roles, manage different outcomes based on data values, or adjust functionality depending on system states.
4. **Error Handling:** They are used to handle errors or exceptional cases gracefully by defining specific actions when certain conditions are encountered, thus improving the robustness of the program.
5. **Code Readability and Maintenance:** Well-structured conditional statements enhance the readability of code by clearly defining how different conditions affect the execution flow. This makes it easier to understand, maintain, and debug the code.
6. **Performance Optimization:** By using conditional statements, developers can optimize performance by avoiding unnecessary computations or operations. For example, a program might skip certain processing steps if specific conditions are met.

Overall, conditional statements are a crucial part of programming that help create flexible, interactive, and efficient applications by allowing the code to make decisions and respond to different situations dynamically.

3.3 BASIC CONDITIONAL STATEMENTS IN JAVA

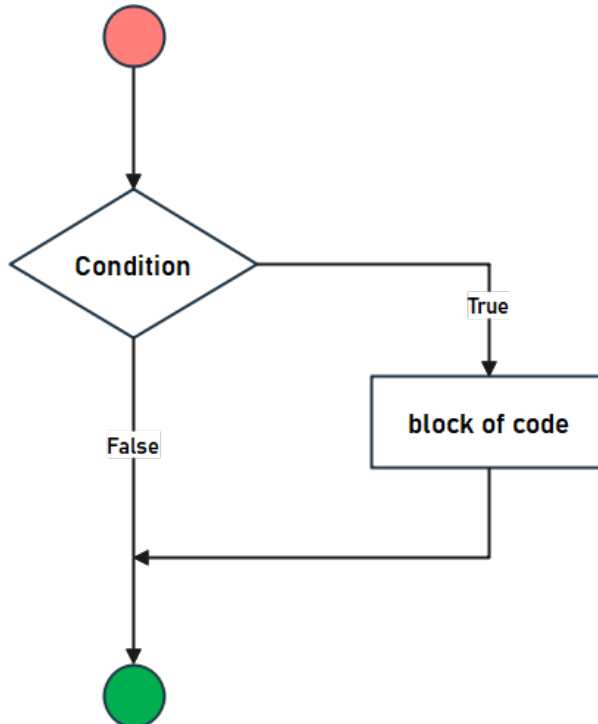
Java provides several basic conditional statements to control the flow of execution based on different conditions. These statements include the if, else if, else, and switch statements. Here's an overview of each:

❖ **if Statement**

The if statement is used to execute a block of code only if a specified condition evaluates to true.

Syntax:

```
if (condition) {  
    // Code to be executed if the condition is true  
}
```

Flowchart:**Fig 3.1. Flow Chart of if Statement****Example:**

```
int age = 18;
if (age >= 18) {
    System.out.println("You are an adult.");
}
```

In this example, the message "You are an adult." will be printed only if the value of age is 18 or greater.

❖ else Statement

The else statement follows an if statement and provides an alternative block of code to execute when the if condition evaluates to false.

Syntax:

```
if (condition) {
    // Code to be executed if the condition is true
} else {
    // Code to be executed if the condition is false
}
```

Flowchart

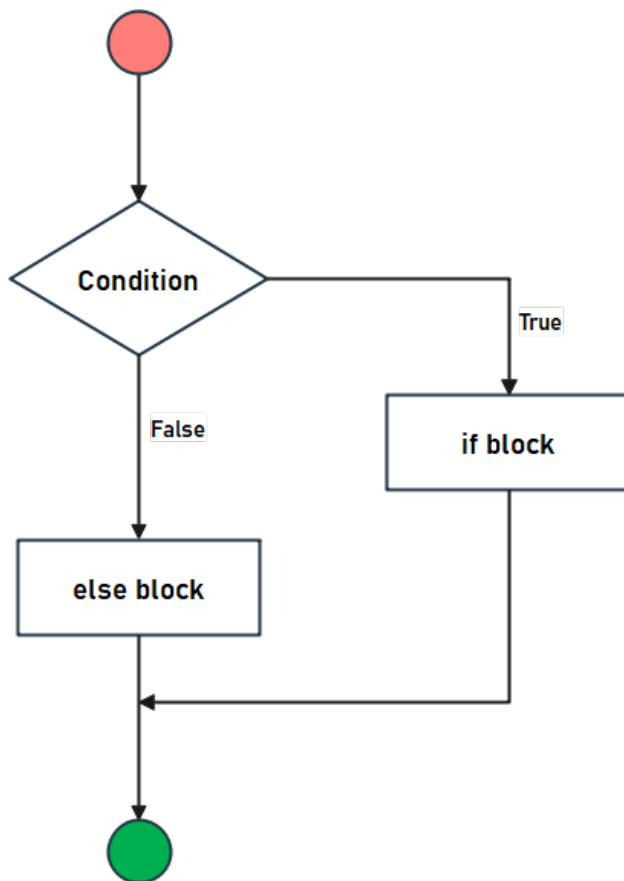


Fig 3.2. Flow Chart of if-else Statement

Example:

```
int age = 16;
if (age >= 18) {
    System.out.println("You are an adult.");
} else {
    System.out.println("You are not an adult.");
}
```

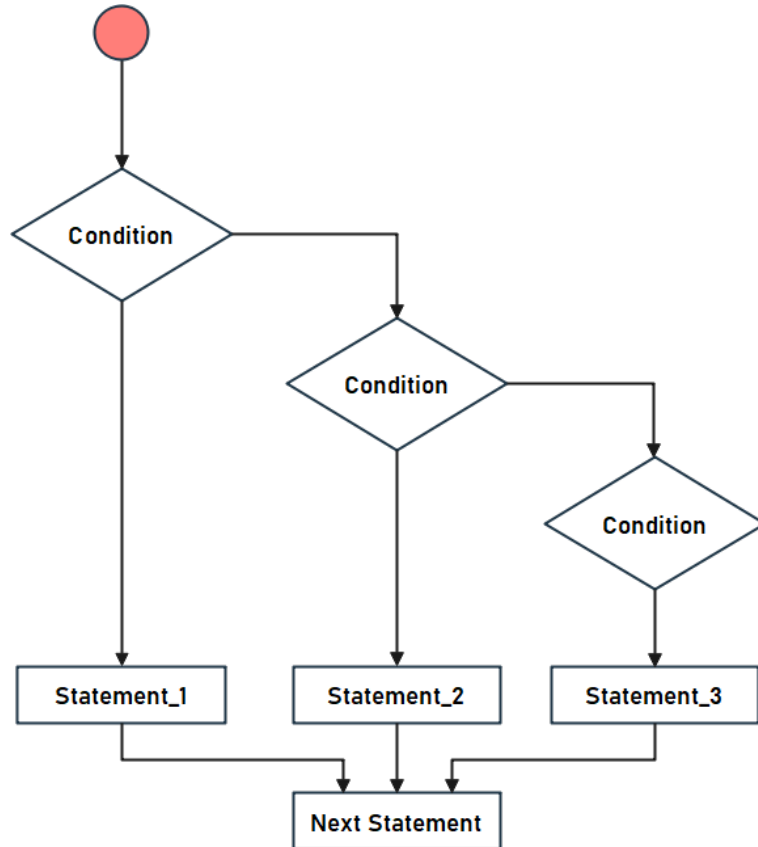
Here, the message "You are not an adult." will be printed since the age is less than 18.

❖ **else if Statement**

The else if statement allows checking multiple conditions sequentially. It is used when you need to check more than two conditions.

Syntax:

```
if (condition1) {
    // Code to be executed if condition1 is true
} else if (condition2) {
    // Code to be executed if condition2 is true
} else {
    // Code to be executed if none of the above conditions are true
}
```

Flowchart:**Fig 3.3. Flow Chart of else-if ladder Statement****Example:**

```
int score = 85;
if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 80) {
    System.out.println("Grade: B");
} else if (score >= 70) {
    System.out.println("Grade: C");
} else {
    System.out.println("Grade: F");
}
```

In this example, the program prints "Grade: B" because the score is between 80 and 89.

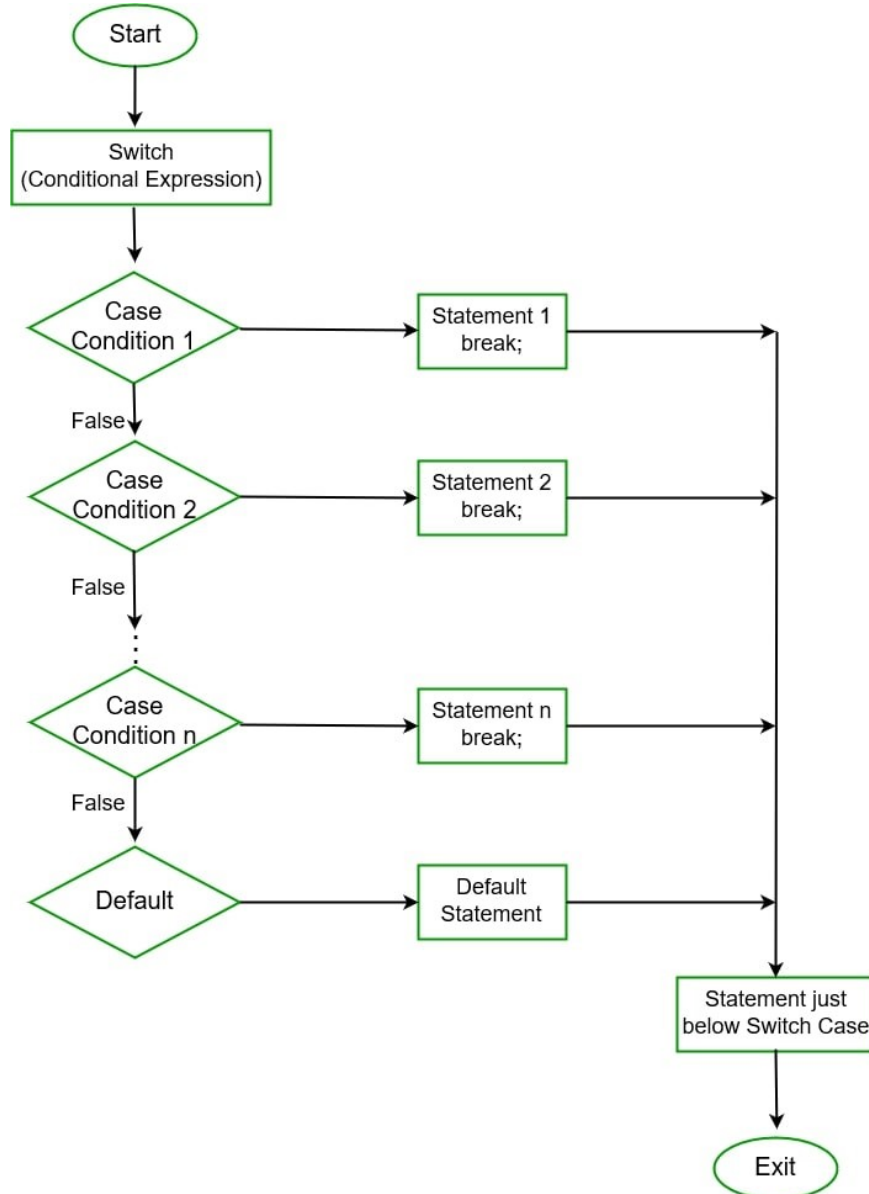
❖ switch Statement

The switch statement allows for selecting one of many code blocks to execute based on the value of an expression. It is generally used when a single variable needs to be compared against multiple possible values.

Syntax:

```
switch (expression) {
    case value1:
        // Code to be executed if expression equals value1
        break;
    case value2:
        // Code to be executed if expression equals value2
```

```
break;  
// More cases as needed  
default:  
    // Code to be executed if no case matches  
}
```

Flowchart:**Fig 3.4 Flow Chart of else-if ladder Statement****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");
}
```

Here, "Wednesday" will be printed because the value of day is 3.

These basic conditional statements provide the foundational building blocks for implementing decision-making logic in Java programs. They enable developers to control the flow of execution based on varying conditions, making their applications more interactive and responsive.

3.4 ADVANCED CONDITIONAL STATEMENTS

In addition to basic conditional statements, Java provides more advanced features to handle complex decision-making scenarios. These include nested if statements, combining conditions, the ternary operator, and enhanced switch statements. Here's a detailed look at each:

❖ Nested if Statements

Nested if statements are used when you need to make a decision within another decision. This allows for more granular control over execution based on multiple layers of conditions.

Syntax:

```
if (condition1) {
    if (condition2) {
        // Code to be executed if both condition1 and condition2 are true
    } else {
        // Code to be executed if condition1 is true but condition2 is false
    }
} else {
    // Code to be executed if condition1 is false
}
```

Example:

```
int age = 20;
boolean hasTicket = true;

if (age >= 18) {
    if (hasTicket) {
        System.out.println("You can enter the movie.");
    } else {
        System.out.println("You need a ticket to enter.");
    }
} else {
    System.out.println("You must be at least 18 years old to enter.");
}
```

In this example, the program checks both the age and whether the person has a ticket, providing a message based on these conditions.

❖ Combining Conditions

Combining conditions allows for more complex decision-making by using logical operators such as && (logical AND), || (logical OR), and ! (logical NOT).

Syntax:

```
if (condition1 && condition2) {  
    // Code to be executed if both conditions are true  
}
```

```
if (condition1 || condition2) {  
    // Code to be executed if at least one condition is true  
}
```

```
if (!condition) {  
    // Code to be executed if the condition is false  
}
```

Example:

```
int temperature = 75;  
boolean isRaining = false;  
  
if (temperature > 70 && !isRaining) {  
    System.out.println("It's a nice day outside.");  
}  
  
if (temperature < 32 || isRaining) {  
    System.out.println("Prepare for cold or wet weather.");  
}
```

Here, the program checks multiple conditions to provide appropriate messages based on the temperature and weather conditions.

3.5 CONTROL FLOW ENHANCEMENTS

Java offers several control flow enhancements that improve the way decisions and branching are handled within a program. These enhancements include the ternary operator, assertions, and advanced features in the switch statement. Here's a detailed look at each enhancement:

❖ Ternary Operator

The ternary operator (? :) is a shorthand for simple if-else statements, often used for concise assignment or return statements. It can be especially useful for reducing verbosity in conditional expressions.

Syntax:

```
result = (condition) ? valueIfTrue : valueIfFalse;
```

Example:

```
int score = 85;  
String grade = (score >= 90) ? "A" : (score >= 80) ? "B" : "C";  
System.out.println("Grade: " + grade);
```

Here, the ternary operator is used to assign a grade based on the score. It's more concise than using nested if-else statements.

❖ Assertions

Assertions are a debugging tool used to test assumptions made in the code. They allow developers to specify conditions that should be true during runtime. If an assertion fails, it throws an `AssertionError`, which can be useful for catching logical errors during development.

Syntax:

```
assert condition : message;
```

Example:

```
int age = 25;
assert age >= 18 : "Age must be 18 or older";
```

To enable assertions, you must run the Java Virtual Machine (JVM) with the `-ea` flag (e.g., `java -ea MyClass`). Assertions are typically used during development and testing rather than in production code.

❖ switch Expression (Java 12+)

Introduced in Java 12, the switch expression enhances the traditional switch statement by allowing it to return values and use a more concise syntax. It also provides improved readability and reduces boilerplate code.

Syntax:

```
result = switch (expression) {
    case value1 -> result1;
    case value2 -> result2;
    default -> defaultResult;
};
```

Example:

```
int day = 3;
String dayName = switch (day) {
    case 1 -> "Monday";
    case 2 -> "Tuesday";
    case 3 -> "Wednesday";
    case 4 -> "Thursday";
    case 5 -> "Friday";
    case 6 -> "Saturday";
    case 7 -> "Sunday";
    default -> "Invalid day";
};
System.out.println(dayName);
```

In this example, the switch expression returns the name of the day based on the value of `day`, with a default case for invalid days.

These control flow enhancements provide more flexibility, readability, and efficiency in handling decision-making and branching in Java programs. They help developers write cleaner and more maintainable code while leveraging the latest language features.

3.6 COMMON USE CASES FOR CONDITIONAL STATEMENTS

Conditional statements are versatile tools in Java programming, used to handle a wide range of scenarios. Here are some common use cases:

❖ Validating User Input

Conditional statements are often used to check and validate user input, ensuring that data meets certain criteria before processing.

Example:

```
import java.util.Scanner;
public class UserInputValidation {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        if (age < 0) {
            System.out.println("Age cannot be negative.");
        } else if (age < 18) {
            System.out.println("You are a minor.");
        } else {
            System.out.println("You are an adult.");
        }
    }
}
```

In this example, the program validates the user's age and provides appropriate messages based on the input.

❖ Implementing Game Logic

Conditional statements are used to create interactive and dynamic game behavior, such as checking win conditions, handling player actions, and managing game states.

Example:

```
public class Game {
    public static void main(String[] args) {
        int playerScore = 95;
        int targetScore = 100;

        if (playerScore >= targetScore) {
            System.out.println("Congratulations! You win!");
        } else {
            System.out.println("Keep trying! Your score is " + playerScore);
        }
    }
}
```

This example checks if the player's score meets or exceeds the target score to determine if they win.

❖ Handling Different Application States

Conditional statements help manage different states of an application, such as loading, running, or error states, by executing different code blocks based on the current state.

Example:

```
public class ApplicationState {
    public static void main(String[] args) {
        String state = "loading";
        switch (state) {
```

```
        case "loading":
            System.out.println("Application is loading...");
            break;
        case "running":
            System.out.println("Application is running.");
            break;
        case "error":
            System.out.println("An error occurred.");
            break;
        default:
            System.out.println("Unknown state.");
    }
}
```

This example uses a switch statement to handle different application states.

❖ Error Handling and Exception Management

Conditional statements are used to handle different error conditions or exceptional cases gracefully, allowing the program to continue running or provide useful feedback.

Example:

```
public class ErrorHandling {
    public static void main(String[] args) {
        int number = -10;

        if (number < 0) {
            System.out.println("Error: Number cannot be negative.");
        } else {
            System.out.println("Number is valid: " + number);
        }
    }
}
```

Here, the program checks for negative numbers and provides an error message if the condition is met.

❖ Control Flow in Loops

Conditional statements inside loops allow for fine-grained control of the loop's execution, such as breaking out of a loop, skipping iterations, or handling specific cases.

Example:

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        System.out.println(i + " is even.");
    } else {
        System.out.println(i + " is odd.");
    }
}
```

In this example, the if-else statement determines whether each number in the loop is even or odd.

❖ **Calculating Discounts or Pricing**

Conditional statements are used to apply different pricing or discounts based on customer eligibility, purchase amounts, or other criteria.

Example:

```
public class Pricing {
    public static void main(String[] args) {
        double purchaseAmount = 150.00;
        double discount;

        if (purchaseAmount >= 100) {
            discount = 0.10; // 10% discount
        } else {
            discount = 0.05; // 5% discount
        }

        double finalPrice = purchaseAmount * (1 - discount);
        System.out.println("Final price after discount: $" + finalPrice);
    }
}
```

This example calculates a discount based on the purchase amount and applies it to the final price.

❖ **User Authentication and Authorization**

Conditional statements help manage user access levels, permissions, and authentication processes in applications.

Example:

```
public class UserAuthentication {
    public static void main(String[] args) {
        String userRole = "admin";

        if (userRole.equals("admin")) {
            System.out.println("Access granted to admin panel.");
        } else if (userRole.equals("user")) {
            System.out.println("Access granted to user dashboard.");
        } else {
            System.out.println("Access denied.");
        }
    }
}
```

Here, the program checks the user's role to determine access rights.

These use cases illustrate how conditional statements are crucial for implementing logic, managing application flow, and handling various scenarios in Java programs.

3.7 BEST PRACTICES

Effective use of conditional statements is essential for writing clean, efficient, and maintainable code. Here are some best practices to follow:

❖ **Avoid Deep Nesting**

Deeply nested conditional statements can make code difficult to read and maintain. To improve readability, try to minimize nesting levels by:

- Using early returns or breaks to exit from a method or loop as soon as a condition is met.
- Refactoring complex conditions into separate methods that return boolean values.

Example:

```
// Avoid deep nesting
if (condition1) {
    if (condition2) {
        if (condition3) {
            // Do something
        }
    }
}
```

```
// Refactored code
if (!condition1) return;
if (!condition2) return;
if (condition3) {
    // Do something
}
```

❖ Use Descriptive Conditionals

Ensure that the conditions in your if, else if, and switch statements are clear and descriptive. Use meaningful variable names and consider using helper methods to encapsulate complex conditions.

Example:

```
// Descriptive conditional
if (user.isEligibleForDiscount()) {
    applyDiscount();
}
```

```
// Less descriptive
if (user.getAge() > 60 && user.hasLoyaltyCard()) {
    applyDiscount();
}
```

❖ Prefer switch for Multiple Conditions

When dealing with a single variable that can have multiple distinct values, using a switch statement can be clearer and more efficient than multiple if-else statements.

Example:

```
// Using switch
switch (dayOfWeek) {
    case MONDAY:
        // Handle Monday
        break;
    case TUESDAY:
        // Handle Tuesday
        break;
    default:
        // Handle other cases
}
```

```
// Using if-else (less preferred)
if (dayOfWeek == MONDAY) {
    // Handle Monday
} else if (dayOfWeek == TUESDAY) {
    // Handle Tuesday
} else {
    // Handle other cases
}
```

❖ Leverage Ternary Operator for Simple Conditions

Use the ternary operator for simple if-else assignments to make the code more concise and readable. Avoid using it for complex conditions or multiple statements.

Example:

```
// Using ternary operator
int max = (a > b) ? a : b;
```

```
// Avoid complex ternary operations
String result = (a > b) ? (a > c ? "a" : "c") : (b > c ? "b" : "c");
```

5. Handle All Possible Cases in switch Statements

Ensure that all possible cases are handled in switch statements, including a default case to manage unexpected values. This helps prevent bugs and ensures robustness.

Example:

```
java
Copy code
switch (status) {
    case ACTIVE:
        // Handle active status
        break;
    case INACTIVE:
        // Handle inactive status
        break;
    default:
        // Handle unexpected status
        System.out.println("Unknown status");
}
```

❖ Use Pattern Matching and switch Expressions (Java 12+ and Java 17+)

Take advantage of advanced features like switch expressions and pattern matching to write more concise and expressive code.

Example with switch expression:

```
String dayName = switch (dayOfWeek) {
    case MONDAY -> "Monday";
    case TUESDAY -> "Tuesday";
    case WEDNESDAY -> "Wednesday";
    default -> "Unknown day";
};
```

Example with pattern matching:

```
Object obj = "Hello";
String result = switch (obj) {
    case String s && s.length() > 5 -> "Long string";
```

```
    case String s -> "Short string";  
    default -> "Not a string";  
};
```

Avoid Overusing Conditional Logic

While conditional logic is powerful, overusing it can lead to complex and hard-to-maintain code. Consider using design patterns, polymorphism, or strategy patterns to handle complex logic in a more manageable way.

Example: Instead of using a lot of if-else statements to handle different behaviors, consider using the Strategy pattern to encapsulate these behaviors.

❖ Document Complex Conditions

When using complex conditions or nested statements, add comments to explain the logic. This will help others (and yourself) understand the intent and functionality of the code.

Example:

```
java
```

```
Copy code
```

```
// Check if user is eligible for a special discount  
if (user.isMember() && user.hasMadePurchaseInLastMonth()) {  
    applySpecialDiscount();  
}
```

By following these best practices, you can write conditional logic that is more readable, maintainable, and effective, improving the overall quality of your Java code.

3.8 SUMMARY

Conditional statements in Java are essential for directing the flow of a program based on varying conditions. These include the if, else if, and else statements, which allow execution of specific code blocks based on whether conditions evaluate to true or false. The switch statement provides a streamlined approach for handling multiple discrete values of a variable. Advanced features like the ternary operator, pattern matching, and enhanced switch expressions further enhance flexibility and readability. By using these constructs, Java developers can implement dynamic decision-making and control the program's execution path effectively.

3.9 TECHNICAL TERMS

- If
- Else
- Switch
- Ternary operator
- Pattern matching
- Condition
- Boolean expression

3.10 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain how you would refactor deeply nested if statements to improve readability and maintainability. Provide a code example.

2. Describe the advantages of using switch expressions introduced in Java 12 over traditional switch statements. Provide an example.
3. Discuss the differences between using the ternary operator and if-else statements for conditional logic. When should each be used?
4. How does pattern matching in switch statements enhance code readability and functionality? Provide an example of pattern matching in use.

Short questions:

1. What is the purpose of the else statement in Java?
2. How do you use the ternary operator in Java?
3. What is the difference between if-else and switch statements?
4. How does the default case work in a switch statement?

3. 11 SUGGESTED READINGS

1. "Java: The Complete Reference" by Herbert Schildt, 12th Edition (2021), McGraw-Hill Education
2. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition (2005), O'Reilly Media
3. "Effective Java" by Joshua Bloch, 3rd Edition (2018), Addison-Wesley Professional

Dr. KAMPA LAVANYA

LESSON- 4

LOOP STATEMENTS

OBJECTIVES

By the end of this chapter, you should be able to:

- execute a block of code multiple times, reducing redundancy and ensuring that repetitive tasks are automated.
- systematically access each element in a collection or array, enabling operations such as processing, searching, or modifying data.
- dynamically control the flow of execution based on conditions, allowing for flexible and adaptive programming.
- handle large datasets or perform calculations repeatedly without manually duplicating code.
- manage and update counters, such as for indexing elements, tracking iterations, or controlling loops.
- These objectives highlight the importance of loop statements in Java for creating efficient, readable, and maintainable code.

STRUCTURE

- 4.1 Introduction
- 4.2 Basic Loop Statements
- 4.3 Nested Loops
- 4.4 Loop Control Statements
- 4.5 Searching and Sorting with Loops
- 4.6 Best Practices
- 4.7 Summary
- 4.8 Technical Terms
- 4.9 Self-Assessment Questions
- 4.10 Suggested Readings

4.1 INTRODUCTION

Loop statements in Java are fundamental constructs that enable developers to execute a block of code repeatedly based on specified conditions. These control structures are essential for automating repetitive tasks, processing collections of data, and managing dynamic execution flows. Java provides several types of loops, including for, while, and do-while, each designed to handle different looping scenarios. By leveraging loops, programmers can efficiently iterate over arrays and collections, implement complex algorithms, and ensure that their code is both concise and maintainable. Understanding how to effectively use these loops is crucial for optimizing performance and achieving flexible and scalable software solutions.

4.2 BASIC LOOP STATEMENTS

In Java, basic loop statements are used to execute a block of code repeatedly based on certain conditions. The primary types of loop statements are for, while, and do-while. Here's a brief overview of each:

❖ for Loop

Definition: The for loop is used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and update.

Syntax:

```
for (initialization; condition; update) {  
    // Code to be executed  
}
```

Flowchart:

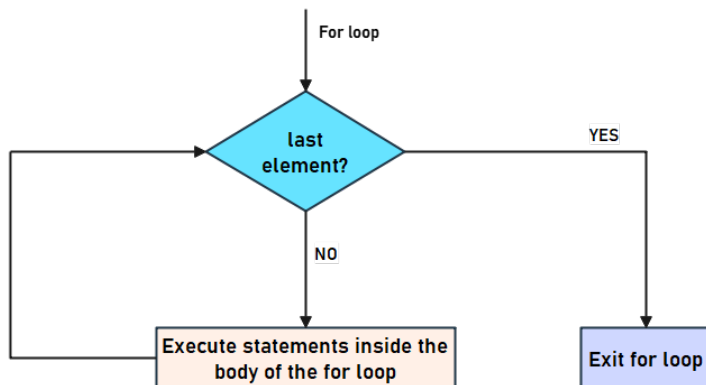


Fig 4.1. Flowchart of For-loop Statement

Example:

```
// Print numbers from 1 to 5  
for (int i = 1; i <= 5; i++) {  
    System.out.println(i);  
}
```

Components:

- Initialization: Sets up the loop control variable.
- Condition: The loop continues as long as this condition is true.
- Update: Modifies the loop control variable after each iteration.

❖ while Loop

Definition: The while loop repeatedly executes a block of code as long as a specified condition is true. It is used when the number of iterations is not known in advance.

Syntax:

```
while (condition) {  
    // Code to be executed  
}
```

Flowchart:

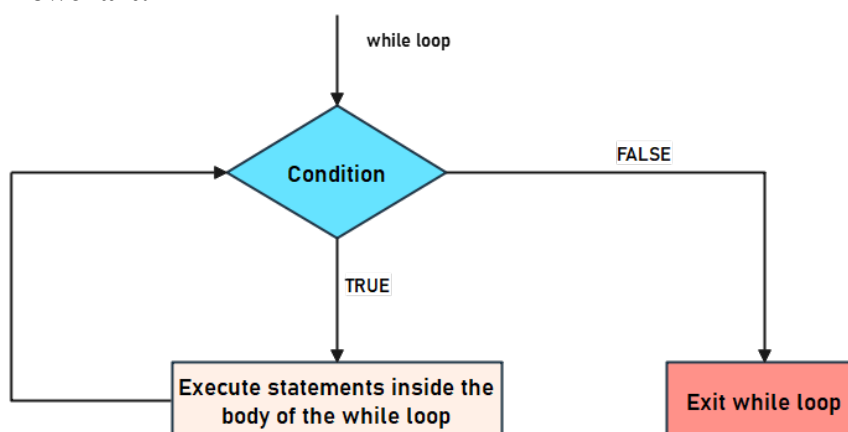


Fig 4.2 Flowchart of While-loop Statement

Example:

```
// Print numbers from 1 to 5
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;
}
```

Components:

- Condition: The loop continues as long as this condition remains true.
- Code Block: Executes each time the condition evaluates to true.
- Update: Typically occurs within the code block to eventually terminate the loop.

❖ do-while Loop

Definition: The do-while loop is similar to the while loop but guarantees that the code block will execute at least once before the condition is tested.

Syntax:

```
do {
    // Code to be executed
} while (condition);
```

Example:

```
// Print numbers from 1 to 5
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <= 5);
```

Components:

- Code Block: Executes first before the condition is tested.
 - Condition: The loop continues as long as this condition is true.
- Summary
- for Loop: Ideal for a known number of iterations.
 - while Loop: Best for when the number of iterations is uncertain.
 - do-while Loop: Ensures the code block executes at least once.

Understanding and using these basic loop constructs allows developers to handle repetitive tasks efficiently and control program flow effectively.

4.3 NESTED LOOPS

These loops in Java refer to the practice of placing one loop inside another loop. This structure is used to perform complex iterations, such as iterating over multi-dimensional arrays or generating patterns. Each loop inside is known as a "nested" loop, and it can be any type of loop (for, while, or do-while).

Syntax for Nested Loops:

Copy code

```
for (initialization; condition; update) {
    for (initialization; condition; update) {
        // Inner loop code
    }
    // Outer loop code
}
```



```
}
```

Examples

1. Printing a Multiplication Table

Example:

```
// Print a multiplication table from 1 to 5
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 5; j++) {
        System.out.print(i * j + "\t"); // Print the product
    }
    System.out.println(); // Move to the next line
}
```

In this example:

- The outer loop (i loop) iterates through the rows.
- The inner loop (j loop) iterates through the columns, printing the product of i and j.

2. Iterating Over a 2D Array

Example:

```
// Define a 2D array
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Print the 2D array
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println(); // Move to the next line
}
```

In this example:

- The outer loop iterates over rows of the 2D array.
- The inner loop iterates over columns of each row.

Common Use Cases

1. **Generating Patterns:** Nested loops are often used to generate patterns or shapes, such as stars or grids.

```
// Print a square pattern of stars
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        System.out.print("* ");
    }
    System.out.println();
}
```

2. **Matrix Operations:** Performing operations on matrices, such as addition, subtraction, or multiplication, often involves nested loops.

3. **Complex Data Structures:** Traversing multi-dimensional data structures or grids.

Performance Considerations

- **Time Complexity:** The time complexity of nested loops is multiplicative. For example, two nested loops each running n times have a time complexity of $O(n^2)$.
- **Efficiency:** Deeply nested loops can lead to performance issues, so it is important to ensure that they are necessary and optimized.

Nested loops are a powerful feature in Java that enable complex iteration and data processing. By understanding their structure and applications, developers can effectively handle multi-dimensional data and create intricate patterns or algorithms.

4.4 LOOP CONTROL STATEMENTS

Loop control statements in Java are used to alter the flow of execution within loops, providing more control over how and when the loops should terminate or continue. The primary loop control statements are `break`, `continue`, and `return`. Here's a detailed look at each:

❖ **break Statement**

Purpose: The `break` statement exits the nearest enclosing loop (`for`, `while`, or `do-while`) and transfers control to the statement immediately following the loop.

Syntax:

```
break;
```

Example:

```
// Find the first number greater than 10 in an array
int[] numbers = {1, 5, 8, 12, 15};
for (int num : numbers) {
    if (num > 10) {
        System.out.println("First number greater than 10: " + num);
        break; // Exit the loop
    }
}
```

The `break` statement exits the `for` loop as soon as a number greater than 10 is found.

❖ **continue Statement**

The `continue` statement skips the current iteration of the nearest enclosing loop and proceeds to the next iteration of the loop.

Syntax:

```
continue;
```

Example:

```
// Print numbers from 1 to 10, skipping multiples of 3
for (int i = 1; i <= 10; i++) {
    if (i % 3 == 0) {
        continue; // Skip the current iteration
    }
    System.out.println(i);
}
```

The `continue` statement skips the printing of numbers that are multiples of 3.

❖ **return Statement**

The `return` statement exits the current method and optionally returns a value. When used within loops, it also exits the method containing the loop.

Syntax:

```
return; // To exit the method without returning a value  
return value; // To exit the method and return a value
```

Example:

```
// Method to find if a number is in an array  
public boolean contains(int[] array, int target) {  
    for (int num : array) {  
        if (num == target) {  
            return true; // Exit the method and return true  
        }  
    }  
    return false; // Return false if target is not found  
}
```

The return statement exits the method as soon as the target value is found in the array, returning true.

Summary

- break: Exits the loop and transfers control to the statement following the loop.
- continue: Skips the rest of the code in the current iteration and proceeds to the next iteration.
- return: Exits the method and optionally returns a value, which can also affect loop execution when used within loops.

These loop control statements provide the flexibility to manage loop execution flow, handle specific conditions, and control how and when loops terminate or skip iterations.

4.5 SEARCHING AND SORTING WITH LOOPS

Searching and sorting are fundamental operations in programming, and loops play a crucial role in implementing these algorithms. Here's a guide to basic searching and sorting techniques using loops in Java:

- ❖ **Searching Algorithms**
- ❖ **Linear Search**

Linear search is a simple algorithm that checks each element in a list or array sequentially until the desired element is found or the end of the list is reached.

Algorithm:

- Iterate through each element of the array.
- Compare the current element with the target value.
- If a match is found, return the index or the element.
- If the end of the array is reached without finding the target, return a failure indicator (e.g., -1).

Example:

```
public class LinearSearch {  
    public static int linearSearch(int[] array, int target) {  
        for (int i = 0; i < array.length; i++) {  
            if (array[i] == target) {  
                return i; // Return index if target is found  
            }  
        }  
        return -1; // Return -1 if target is not found  
    }  
}
```

```
}  
  
public static void main(String[] args) {  
    int[] numbers = {3, 5, 7, 9, 11};  
    int index = linearSearch(numbers, 7);  
    System.out.println("Index of 7: " + index);  
}  
}
```

❖ **Sorting Algorithms**

❖ **Bubble Sort**

Bubble sort is a straightforward sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

Algorithm:

- Iterate through the array.
- Compare each pair of adjacent elements.
- Swap them if they are in the wrong order.
- Repeat the process until no swaps are needed.

Example:

```
public class BubbleSort {  
    public static void bubbleSort(int[] array) {  
        int n = array.length;  
        for (int i = 0; i < n - 1; i++) {  
            for (int j = 0; j < n - 1 - i; j++) {  
                if (array[j] > array[j + 1]) {  
                    // Swap elements  
                    int temp = array[j];  
                    array[j] = array[j + 1];  
                    array[j + 1] = temp;  
                }  
            }  
        }  
    }  
}  
  
public static void main(String[] args) {  
    int[] numbers = {64, 34, 25, 12, 22};  
    bubbleSort(numbers);  
    System.out.println("Sorted array: " + Arrays.toString(numbers));  
}  
}
```

❖ **Selection Sort**

Selection sort is a simple sorting algorithm that divides the array into two parts: a sorted part and an unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the end of the sorted part.

Algorithm:

- Iterate through the array.
- Find the minimum (or maximum) element in the unsorted part.
- Swap it with the first unsorted element.
- Move the boundary between the sorted and unsorted parts.

Example:

```
public class SelectionSort {
    public static void selectionSort(int[] array) {
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }
            // Swap the found minimum element with the first unsorted element
            int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
    }

    public static void main(String[] args) {
        int[] numbers = {64, 34, 25, 12, 22};
        selectionSort(numbers);
        System.out.println("Sorted array: " + Arrays.toString(numbers));
    }
}
```

❖ Insertion Sort

Insertion sort builds the final sorted array one item at a time. It takes each element from the input and inserts it into its correct position within the already sorted portion of the array.

Algorithm:

- Iterate through the array from the second element to the last.
- For each element, compare it to the elements in the sorted portion and insert it into its correct position.

Example:

```
public class InsertionSort {
    public static void insertionSort(int[] array) {
        int n = array.length;
        for (int i = 1; i < n; i++) {
            int key = array[i];
            int j = i - 1;
            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j--;
            }
            array[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] numbers = {64, 34, 25, 12, 22};
        insertionSort(numbers);
    }
}
```

```

        System.out.println("Sorted array: " + Arrays.toString(numbers));
    }
}

```

Summary

- **Linear Search:** Simple and effective for small datasets or unsorted arrays.
- **Bubble Sort:** Easy to implement but inefficient for large datasets due to its $O(n^2)$ complexity.
- **Selection Sort:** Straightforward but also has $O(n^2)$ complexity; useful for small arrays.
- **Insertion Sort:** More efficient than bubble and selection sorts for small or partially sorted arrays.

Table 4.1 Differences between linear and binary search

Linear search	Binary search
<ul style="list-style-type: none"> • In linear search, input data doesn't need to be sorted . 	<ul style="list-style-type: none"> • Whereas, in binary search, input data has to be sorted according to the order.
<ul style="list-style-type: none"> • It is also referred as sequential search. 	<ul style="list-style-type: none"> • It is also referred to as half-interval search.
<ul style="list-style-type: none"> • The time complexity of the linear search is $O(n)$ 	<ul style="list-style-type: none"> • The time complexity of the binary search is $O(\log n)$
<ul style="list-style-type: none"> • Multi-dimensional array is used for linear search. 	<ul style="list-style-type: none"> • A single dimensional array is used for linear search.
<ul style="list-style-type: none"> • It operates equality comparisons 	<ul style="list-style-type: none"> • Binary search operates ordering comparisons
<ul style="list-style-type: none"> • Linear search is less complex and involves a slow process 	<ul style="list-style-type: none"> • Binary search is more complex and has a fast process

These algorithms illustrate the use of loops to perform common operations and help in understanding how basic data manipulation tasks are carried out in programming.

4.6 BEST PRACTICES

When working with loop statements in Java, following best practices can help ensure that your code is efficient, readable, and maintainable. Here are some key best practices for using loops effectively:

❖ **Avoid Infinite Loops**

Ensure that your loop has a well-defined exit condition to avoid infinite loops.

Example:

```

// Infinite loop example (avoid this)
while (true) {
    // Some code
}

// Proper loop with exit condition
int i = 0;

```

```
while (i < 10) {  
    // Some code  
    i++;  
}
```

Always ensure that the loop condition will eventually become false. Consider using loop control variables and updates within the loop body to prevent infinite execution.

❖ Use Descriptive Loop Variables

Use meaningful names for loop control variables to make your code more readable and self-explanatory.

Example:

```
// Less descriptive variable name  
for (int i = 0; i < array.length; i++) {  
    System.out.println(array[i]);  
}
```

```
// More descriptive variable name  
for (int index = 0; index < array.length; index++) {  
    System.out.println(array[index]);  
}
```

Descriptive names help others (and yourself) understand the purpose of the loop variable, improving code readability.

❖ Optimize Loop Performance

Optimize loops to avoid unnecessary computations or operations within the loop.

Example:

```
// Inefficient example  
for (int i = 0; i < array.length; i++) {  
    for (int j = 0; j < array.length; j++) {  
        // Some operations  
    }  
}
```

```
// More efficient example (if array.length does not change)  
int length = array.length;  
for (int i = 0; i < length; i++) {  
    for (int j = 0; j < length; j++) {  
        // Some operations  
    }  
}
```

Calculating the length of an array or collection once before the loop can reduce redundant operations and improve performance.

❖ Minimize Nested Loops

Avoid deep nesting of loops when possible. Deeply nested loops can lead to performance issues and complex code.

Example:

```
// Deeply nested loops (use with caution)  
for (int i = 0; i < 10; i++) {  
    for (int j = 0; j < 10; j++) {
```

```
    for (int k = 0; k < 10; k++) {  
        // Some operations  
    }  
}
```

// Alternative approach

// Use simpler logic if possible or break down into functions

Reducing the depth of nested loops can simplify your code and make it easier to understand. Look for opportunities to optimize or refactor complex loop structures.

❖ Use Loop Control Statements Wisely

Use break and continue statements judiciously to control loop execution and improve readability.

Example:

// Using break to exit early

```
for (int i = 0; i < array.length; i++) {  
    if (array[i] == target) {  
        System.out.println("Target found!");  
        break;  
    }  
}
```

// Using continue to skip iterations

```
for (int i = 0; i < array.length; i++) {  
    if (array[i] % 2 == 0) {  
        continue; // Skip even numbers  
    }  
    System.out.println(array[i]); // Process odd numbers  
}
```

break and continue can help manage loop flow effectively, but excessive use can make code harder to follow. Ensure their usage is clear and purposeful.

❖ Avoid Unnecessary Computations

Avoid placing computationally expensive operations or function calls inside the loop condition or body if they do not need to be executed repeatedly.

Example:

// Inefficient example

```
for (int i = 0; i < array.length; i++) {  
    if (array[i] < computeExpensiveValue()) {  
        // Some operations  
    }  
}
```

// More efficient example

```
int expensiveValue = computeExpensiveValue();  
for (int i = 0; i < array.length; i++) {  
    if (array[i] < expensiveValue) {  
        // Some operations  
    }  
}
```



```
}
```

Compute values outside the loop if they do not change, and reuse the result within the loop to improve efficiency.

❖ **Ensure Proper Resource Management**

When dealing with resources like files or network connections, ensure that resources are properly managed and closed, typically using try-with-resources or finally blocks.

Example:

```
// Proper resource management with try-with-resources
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        // Process each line
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Ensuring resources are closed properly prevents resource leaks and potential issues with file or network operations.

❖ **Test with Edge Cases**

Test your loops with various input scenarios, including edge cases such as empty arrays, single-element arrays, or large datasets.

Example:

```
// Test edge cases
int[] emptyArray = {};
int[] singleElementArray = {1};
int[] largeArray = new int[10000]; // Large dataset
```

```
// Run tests for these scenarios
```

Testing with different scenarios ensures that your loops handle various input conditions correctly and robustly.

Following these best practices helps you write efficient, readable, and maintainable loop-based code. By avoiding common pitfalls and optimizing loop performance, you can improve the quality of your Java applications and ensure that they function as intended.

4.7 SUMMARY

Loop statements in Java are essential control structures that allow developers to execute a block of code multiple times, facilitating tasks such as iteration over data, repetitive processing, and dynamic control flow. Java offers three primary types of loops: for, while, and do-while. The for loop is best suited for scenarios with a known number of iterations, while the while loop is ideal for cases where the number of iterations is uncertain, and the do-while loop guarantees at least one execution of the loop body. Nested loops enable handling complex data structures like multi-dimensional arrays and generating intricate patterns. Best practices for using loops include avoiding infinite loops, optimizing performance, and minimizing deep nesting. Effective use of loop control statements (break, continue, and return) further enhances loop management. By adhering to these principles, developers can

write efficient, maintainable, and robust loop constructs in Java, addressing a wide range of programming challenges.

4.8 TECHNICAL TERMS

- while
- for
- do-while
- nested loop
- break
- continue
- return

4.9 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain the structure and usage of a for loop in Java, and provide an example.
2. Describe the use of nested loops in Java and provide an example where nested loops are necessary.
3. Discuss how you would handle performance optimization when using loops in Java. Provide an example.
4. Explain the difference between using a while loop and a do-while loop with an example. When would you prefer one over the other?
5. How do loop control statements like break and continue affect loop execution? Provide examples of their use.

Short questions:

1. What is a for loop in Java?
2. How does a while loop differ from a do-while loop in Java?
3. What is the purpose of the break statement in a loop?
4. What does the continue statement do in a loop?
5. How can you exit a loop early based on a condition?

4.10 SUGGESTED READINGS

1. "Java: The Complete Reference" by Herbert Schildt, 12th Edition (2021), McGraw-Hill Education
2. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition (2005), O'Reilly Media
3. "Effective Java" by Joshua Bloch, 3rd Edition (2018), Addison-Wesley Professional

Dr. KAMPA LAVANYA

LESSON- 5

ARRAYS and STRINGS

OBJECTIVES:

After going through this lesson, you will be able to

- Learn about different types of arrays
- Understand how to determine the length of an array
- Explore various operations on arrays
- Learn the different ways to create strings
- Understand the concept of string immutability
- Explore the impact of string operations on performance.

STRUCTURE:

- 5.1 Arrays – Introduction
- 5.2 Types of Array
- 5.3 Creation of Arrays
 - 5.3.1. Single-Dimensional Arrays
 - 5.3.1.1 Characteristics of One-Dimensional Arrays
 - 5.3.2. Multi-Dimensional Arrays
 - 5.3.2.1 Two-Dimensional Arrays
- 5.4 Operation performed on Arrays
 - 5.4.1. Modifying Elements:
 - 5.4.2. Traversing an Array
 - 5.4.3. Finding the Length of an Array
 - 5.4.4. Copying an Array
 - 5.4.5. Sorting an Array
 - 5.4.6. Accessing an element
 - 5.4.7. Traversing a 2D array
 - 5.4.8. Searching an Element
- 5.5 Strings – Introduction
- 5.6 Creating Strings
 - 5.6.1. Using String Literals
 - 5.6.2. Using the 'new' Keyword
 - 5.6.3. Using Character Arrays
- 5.7 String class methods
- 5.8 String comparison
- 5.9 Immutability of Strings
- 5.10 Summary
- 5.11 Technical Terms
- 5.12 Self-Assessment Questions
- 5.13 Further Readings

5.1 ARRAYS – INTRODUCTION

Java arrays are a fundamental data structure that allows developers to store multiple values of the same type in a single, contiguous block of memory. An array in Java is a collection of similar data types, and it is indexed, meaning each element in the array is

identified by a specific number, known as an index. Arrays are widely used in Java programming for various tasks, including storing lists of items, manipulating data, and implementing algorithms that require data storage and retrieval. Understanding arrays is essential for any Java programmer as they provide the foundation for more complex data structures and algorithms.

In Java, arrays are objects that are dynamically allocated on the heap. The size of an array is fixed at the time of its creation, which means that once an array is created, it cannot grow or shrink. This fixed size is both a strength and a limitation of arrays. On the one hand, it allows for efficient memory management since the memory required for an array is allocated in one go, making access to its elements fast and predictable. On the other hand, it means that if you need to add more elements than the array can hold, you'll need to create a new array with a larger size and copy the elements over.

There are different types of arrays in Java, including single-dimensional and multi-dimensional arrays. A single-dimensional array is the simplest form of an array, which can be thought of as a list of elements, all of the same type. Multi-dimensional arrays, on the other hand, are arrays of arrays. The most common type of multi-dimensional array is the two-dimensional array, which can be visualized as a grid or table of elements. These multi-dimensional arrays are useful for representing more complex data structures, such as matrices or graphs.

In addition to their basic functionality, Java arrays come with a host of utility functions provided by the `java.util.Arrays` class. This class includes static methods that can perform tasks such as sorting arrays, filling arrays with a specific value, copying arrays, and converting arrays to strings. These utilities make working with arrays in Java more flexible and powerful, allowing developers to handle arrays in a more sophisticated and streamlined manner. As such, arrays are not just a collection of elements but are a robust tool for managing and manipulating data in Java.

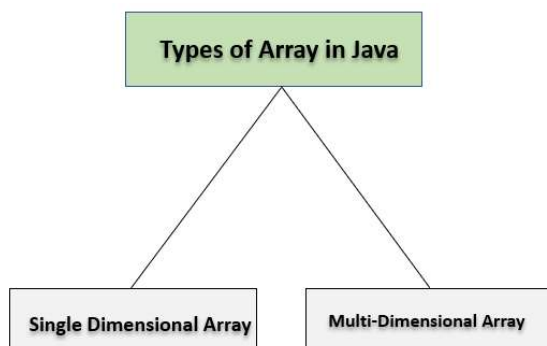
5.2 TYPES OF ARRAY

In Java, arrays can be categorized into several types based on their dimensions and structure.

A one-dimensional array in java is **an object**. They are dynamically created and may be assigned to variables of type Object. An Object class in java is the parent class of all classes by default. All the methods in the class Object can be invoked on an array.

There are two types of array:

- One-dimensional array
- Multi-dimensional array



5.2 types of arrays in Java

5.3 CREATION OF ARRAYS

5.3.1. Single-Dimensional Arrays

A one-dimensional array, often referred to as a vector, is a data structure that stores a sequence of elements of the same type in a contiguous block of memory. Here's a breakdown of its key characteristics:

5.3.1.1 Characteristics of One-Dimensional Arrays:

1. **Single Index Access:** Each element in the array can be accessed using a single index. For example, in an array `arr`, `arr[0]` accesses the first element, `arr[1]` accesses the second element, and so on.
2. **Fixed Size:** The size of a one-dimensional array is determined when it is created and cannot be changed. This means the number of elements it can hold is fixed.
3. **Contiguous Memory:** The elements of the array are stored in contiguous memory locations. This makes accessing elements very efficient because you can directly compute the memory address of any element based on its index.
4. **Homogeneous Elements:** All elements in the array must be of the same data type, such as integers, floats, characters, or any user-defined type

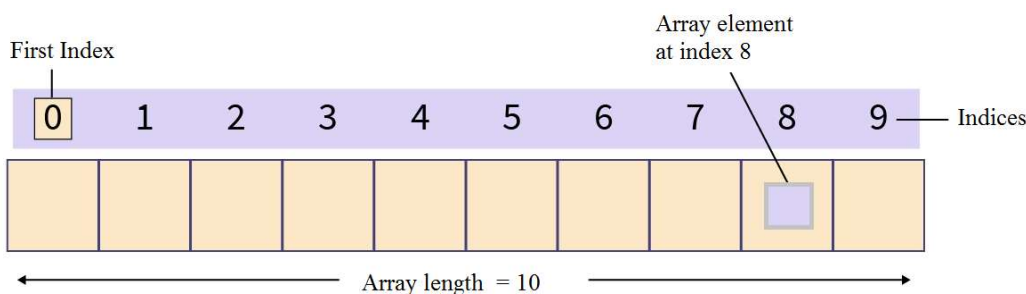


Figure 5.2 Single - dimensional array representation

Syntax:

```
dataType[ ] arrayName; // Declaration
arrayName = new dataType[arraySize]; // Instantiation
// or
dataType[] arrayName = new dataType[arraySize]; // Declaration and
instantiation
```

Example:

```
int[] numbers = new int[5]; // Creates an array of integers with 5 elements
numbers[0] = 10; // Assigns the value 10 to the first element
numbers[1] = 20; // Assigns the value 20 to the second element
```

```
// Declaring, instantiating, and initializing an array in one line
String[] students = {"Shourya "Arya", "Surya"};
System.out.println(students[0]); // Outputs "Arya"
```

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use `new`.

For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.
class AutoArray
{
    public static void main(String args[])
    {
        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
        System.out.println("April has " + month_days[3] + " days.");
    }
}
```

Example Program 1: Java Program to Illustrate Wrong Way of Copying an Array

// A Java program to demonstrate that simply assigning one array reference is incorrect

```
public class Sample {
    public static void main(String[] args)
    {
        int a[] = { 56, 34, 32 };

        // Create an array b[] of same size as a[]
        int b[] = new int[a.length];

        // Doesn't copy elements of a[] to b[], only makes b refer to same location
        b = a;

        // Change to b[] will also reflect in a[] as 'a' and 'b' refer to same location.
        b[0]++;

        System.out.println("Contents of a[] ");
        for (int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");

        System.out.println("\n\nContents of b[] ");
        for (int i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
    }
}
```

Output:

Contents of a[]

56 34 32

Contents of b[]

56 34 32

Example program 2:

```
import java.util.Arrays;
class SrtAry {
    public static void main(String args[])
    {
        int[] arr = { 25, -34, 45, 78, 99, -51, 230 };
    }
}
```

```
System.out.println("The original array is: ");
for (int num : arr) {
    System.out.print(num + " ");
}
Arrays.sort(arr);
System.out.println("\nThe sorted array is: ");
for (int num : arr) {
    System.out.print(num + " ");
}
}
```

The original array is:

25, -34, 45, 78, 99, -51, 230

The sorted array is:

-42 -2 5 7 23 87 509

5.3.2. Multi-Dimensional Arrays

Multi-dimensional arrays are arrays that contain other arrays as their elements. The most common form is the two-dimensional array, which can be visualized as a table or grid. However, Java supports arrays with more than two dimensions.

In Java, multidimensional arrays are actually arrays of arrays. These, as we might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences.

5.3.2.1 Two-Dimensional Arrays

A two-dimensional array in Java is essentially an array of arrays. It is often used to represent matrices, tables, or grids.

To declare a multidimensional array variable, specify each additional index using another set of square brackets.

Conceptually, this array will look like the one shown in Figure 5.2

Syntax:

```
dataType[ ][ ] arrayName; // Declaration
arrayName = new dataType[rows][columns]; // Instantiation
// or
dataType[ ][ ] arrayName = new dataType[rows][columns]; // Declaration and instantiation
```

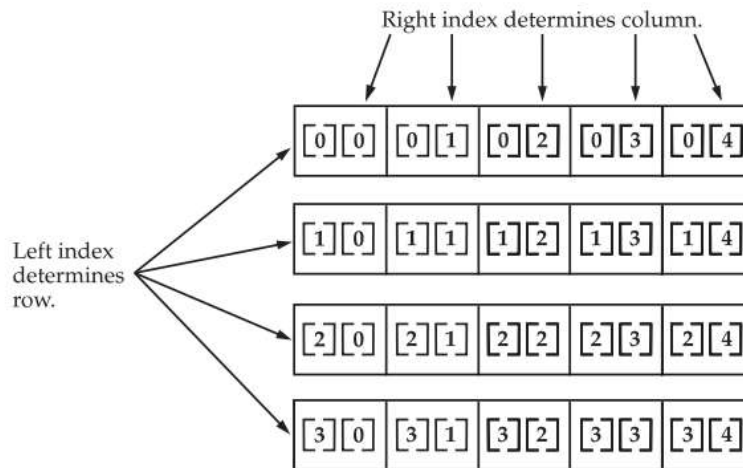


Figure 5.3 structure of multi-dimensional array

For example, the following declares a twodimensional array variable called twoD.

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int..

More examples:

```
int[][] matrix = new int[3][3]; // Creates a 3x3 matrix
```

```
matrix[0][0] = 1; // Assigns 1 to the element at first row, first column
```

```
matrix[0][1] = 2; // Assigns 2 to the element at first row, second column
```

// declaring, instantiating, and initializing a 2D array in one line

```
int[][] table = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

```
System.out.println(table[1][2]);
```

The above code outputs 6 (second row, third column)

Each type of array in Java serves different use cases depending on the data structure requirements. Single-dimensional arrays are straightforward and useful for simple lists, multi-dimensional arrays are excellent for representing grid-like structures, and jagged arrays offer flexibility when dealing with non-uniform data.

Example programs:

Write a java program to perform addition on two matrices

```
public class MatAdd
```

```
{
```

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

```
    {
```

```
        //creating two matrices
```

```
        int a[][]={{1,3,4},{2,4,3},{3,4,5}};
```

```
        int b[][]={{1,3,4},{2,4,3},{1,2,4}};
```

```
        //creating another matrix to store the sum of two matrices
```

```
        int c[][]=new int[3][3]; //3 rows and 3 columns
```



```

//adding and printing addition of 2 matrices
for(int i=0;i<3;i++)
{
    for(int j=0;j<3;j++)
    {
        c[i][j]=a[i][j]+b[i][j]; //use - for subtraction
        System.out.print(c[i][j]+" ");
    }
    System.out.println();//new line
}
}
}

```

5.4 OPERATION PERFORMED ON ARRAYS

The following are some common operations that can be performed on arrays in Java

5.4.1 Modifying Elements:

We can modify an element in an array by directly assigning a new value to a specific index.

Example:

```
int[] ages = {10, 20, 30, 40, 50};
```

```
// Modifying the second element
```

```
ages[1] = 25;
```

```
// Printing the modified array
```

```
System.out.println(ages[1]); // Output: 25
```

```
...
```

5.4.2 Traversing an Array

Traversing an array means accessing each element of the array one by one. This can be done using a `for` loop or an enhanced `for` loop (also known as the "for-each" loop).

Example:

```
int[] ages = {10, 20, 30, 40, 50};
```

```
// Using a traditional for loop
```

```
for (int i = 0; i < ages.length; i++) {
    System.out.println(ages[i]);
}
```

```
// Using an enhanced for loop
```

```
for (int age : ages) {
    System.out.println(age);
}
```

5.4.3 Finding the Length of an Array

The length of an array (number of elements it can hold) can be found using the `length` property.

Example:

```
int[] ages = {10, 20, 30, 40, 50};
```

```
// Finding the length of the array
```

```
System.out.println("The length of the array is: " + ages.length);
```

Output: 5

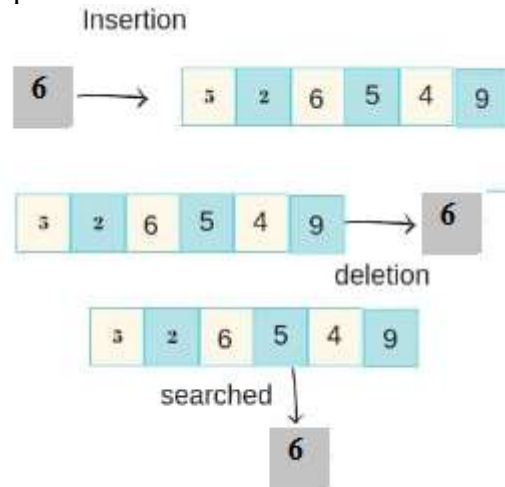


Figure 5.4 structure of multi-dimensional array

5.4.4 Copying an Array

We can copy elements from one array to another using the `System.arraycopy()` method or `Arrays.copyOf()` from the `java.util.Arrays` class.

Example:

```
int[] source = {1, 2, 3, 4, 5};
int[] destination = new int[5];
```

```
// Copying elements using System.arraycopy()
System.arraycopy(source, 0, destination, 0, source.length);
```

```
for (int num : destination) {
    System.out.println(num);
}
```

5.4.5 Sorting an Array

Java provides a built-in method to sort arrays using the `Arrays.sort()` method from the `java.util.Arrays` class.

Example:

```
import java.util.Arrays;
```

```
int[] numbers = {5, 2, 8, 3, 1};
```

```
// Sorting the array
Arrays.sort(numbers);
```

```
for (int number : numbers) {
    System.out.println(number);
}
```

Output: 1 2 3 5 8

5.4.6 Accessing an element

Accessing an element from a multidimensional array in Java is done using multiple indices, one for each dimension.

```
System.out.println(matrix[1][2]);
```

- `matrix[1]` refers to the second row (index 1) of the array.
- `matrix[1][2]` refers to the third element (index 2) in the second row.

Output: 5

5.4.7 Traversing a 2D array

The process of accessing an individual element can be used during a 2D array traversal, which can be used to return all elements in a 2D array.

For a 2D array, this initial traversal is used to access each array in the 2D array, and a nested for loop is needed to access each element within the selected array:

```
for (int i = 0; i < matrix.length; i++) {  
    for (int j = 0; j < matrix[i].length; j++) {  
        System.out.print(matrix[i][j] + " ");  
    }  
    System.out.println();  
}
```

5.4.8 Searching an Element

We can search for an element in an array using a loop or using built-in methods like `Arrays.binarySearch()`.

Example:

```
import java.util.Arrays;
```

```
int[] numbers = {1, 2, 3, 4, 5};
```

```
// Searching for an element (binary search requires the array to be sorted)  
int index = Arrays.binarySearch(numbers, 3);
```

```
if (index >= 0) {  
    System.out.println("Element found at index: " + index);  
} else {  
    System.out.println("Element not found");  
}
```

5.5 STRINGS – INTRODUCTION

In Java, a 'String' is a sequence of characters. It is one of the most commonly used data types and is implemented as a class ('`java.lang.String`'). Strings are immutable, meaning once a 'String' object is created, it cannot be changed. This immutability offers benefits such as thread safety and efficient memory usage.

5.6 CREATING STRINGS

There are several ways to create strings in Java:

5.6.1 Using String Literals: When a string is created using string literals, it is stored in the string pool. If the same string literal is used again, Java reuses the string from the pool instead of creating a new object.

Example:

```
String str1 = "Hello, World!";
```

5.6.2 Using the 'new' Keyword: This creates a new string object, bypassing the string pool.

Example:

```
String str2 = new String("Acharya Nagarjuna University")
```

5.6.3 Using Character Arrays: A string can be created from a character array using the 'String' constructor.

Example:

```
char[] charArray = {'A', 'c', 'h', 'a', 'r', 'y', 'a'};  
String str3 = new String(charArray);
```

5.7 STRING CLASS METHODS

The 'String' class provides many useful methods for manipulating and working with strings. Here are some common ones:

1. length(): Returns the length of the string.

```
String str = "University";  
int len = str.length();
```

Output: 9

2. charAt(int index): Returns the character at the specified index.

```
char ch = str.charAt(1);
```

Output: 'n'

3. substring(int beginIndex, int endIndex): Returns a substring from the specified 'beginIndex' to 'endIndex'.

```
String substr = str.substring(1, 4);
```

Output: "niv"

4. indexOf(String str): Returns the index of the first occurrence of the specified substring.

```
int index = str.indexOf("v");
```

Output: 3

5. toLowerCase() and toUpperCase(): Converts all characters in the string to lowercase or uppercase.

```
String lower = str.toLowerCase();
```

Output: "university "

```
String upper = str.toUpperCase();
```

Output: "UNIVERSITY"

6. trim(): Removes leading and trailing whitespace.

```
String strWithSpaces = " Surya ";  
String trimmedStr = strWithSpaces.trim();  
Output: "Surya"
```

7. 'replace(char oldChar, char newChar)': Replaces occurrences of a specified character with another character.

```
String replacedStr = str.replace('t', 'Z');  
Output: " UniversiZy "
```

8. equals(Object obj): Compares the string to another object for equality.

```
String str2 = "Hello";  
boolean isEqual = str.equals(str2);  
Output: true
```

9. equalsIgnoreCase(String anotherString): Compares two strings, ignoring case considerations.

```
String str3 = "university";  
boolean isEqualIgnoreCase = str.equalsIgnoreCase(str3);  
Output: true
```

10. split(String regex): Splits the string around matches of the given regular expression.

```
String sentence = "Acharya Nagarjuna University";  
String[] words = sentence.split(" ");  
Output: ["Acharya", "Nagarjuna", "University"];
```

5.8 STRING COMPARISON

1. equals(): Compares two strings for content equality.

```
String str1 = "Hello";  
String str2 = "Hello";  
boolean result = str1.equals(str2);  
Output: true
```

2. equalsIgnoreCase(): Compares two strings for equality, ignoring case.

```
String str3 = "hello";  
boolean result = str1.equalsIgnoreCase(str3);  
Output: true
```

3. compareTo(): Compares two strings lexicographically.

The 'compareTo' method returns:

- '0' if the strings are equal
- A positive number if the first string is lexicographically greater
- A negative number if the first string is lexicographically smaller

Example:

```
String str4 = "apple";  
String str5 = "banana";  
int comparison = str4.compareTo(str5);  
Output: Negative number (-1)
```

4. '==' operator: Compares references, not values. It checks if two strings point to the same memory location.

```
String str6 = new String("Hello");  
boolean isSameReference = (str1 == str6);  
Output: false
```

5.9 IMMUTABILITY OF STRINGS

Strings in Java are immutable. This means once a 'String' object is created, its value cannot be changed. Any modification to a string results in the creation of a new string object.

Why Strings Are Immutable:

- 1. Security:** Strings are frequently used as parameters in network connections, file paths, etc. Immutable strings ensure that these values cannot be changed once created, reducing security risks.
- 2. Synchronization and Concurrency:** Immutable strings are inherently thread-safe since their values cannot change after creation. This eliminates the need for synchronization when multiple threads are working with strings.
- 3. Performance:** Java's string pool reuses immutable string literals, which saves memory and reduces the overhead of creating new string objects.

Example 1:

```
String str = "Hello";  
str.concat(" World"); // The original string 'str' is not modified  
System.out.println(str);  
Output: "Hello"
```

Example 2:

```
String newStr = str.concat(" World");  
System.out.println(newStr);  
Output: "Hello World"
```

In this example, 'str.concat(" World")' does not modify 'str'. Instead, it creates a new string "Hello World" and returns it. The original string 'str' remains unchanged, demonstrating the immutability of strings in Java.

5.10 SUMMARY

The chapters on Arrays and Strings in Java cover essential concepts and operations for handling these fundamental data types. Arrays are collections of elements of the same type, and Java supports both single-dimensional and multi-dimensional arrays. Key topics include array declaration, initialization, accessing elements, finding an array's length, and various manipulations like sorting and searching. The array name is a reference to the memory location where the array is stored, and the length property gives the number of elements in the array. Strings in Java are sequences of characters that are immutable, meaning once created, their content cannot be changed. Topics covered include creating strings using literals and constructors, various String class methods for manipulation and inspection, comparing strings using methods like equals() and compareTo(), and understanding the immutability of strings, which provides benefits like thread safety and efficient memory usage through the string pool.

5.11 TECHNICAL TERMS

Java Array, single – dimension, multi – dimension, String, Immutability

5.12 SELF ASSESSMENT QUESTIONS

Essay questions:

1. What is an array in Java, and how is it different from a single variable?
2. How do you find the length of an array in Java? Provide a code example.
3. Explain the difference between == and equals() when comparing strings in Java.
4. Give an example of how to declare and initialize a multi-dimensional array in Java.

Short Answer Questions:

1. Discuss the advantages and disadvantages of using arrays in Java. Include examples to illustrate your points.
2. Write a java program to perform multiplication on two matrices
3. Explain the concept of immutability in Java strings. How does this feature benefit Java programs, and what are some potential drawbacks?
4. Examine the role of the length property in arrays and strings in Java. How does it differ between the two, and why is this distinction important?

5.13 SUGGESTED READINGS

- 1) Herbert Schildt and Dale Skrien “Java Fundamentals –A comprehensive Introduction”, McGraw Hill, 1st Edition, 2013.
- 2) Herbert Schildt, “Java the complete reference”, McGraw Hill, Osborne, 11th Edition, 2018.
- 3) T. Budd “Understanding Object-Oriented Programming with Java”, Pearson Education, Updated Edition (New Java 2 Coverage), 1999 REFERENCE BOOKS:
- 4) P.J. Dietel and H.M. Dietel “Java How to program”, Prentice Hall, 6th Edition, 2005.
- 5) P. Radha Krishna “Object Oriented programming through Java”, CRC Press, 1st Edition, 2007.
- 6) Malhotra and S. Choudhary “Programming in Java”, Oxford University Press, 2nd Edition, 2014

Dr. U. SURYA KAMESWARI

LESSON- 6

CLASSES and OBJECTS

OBJECTIVES:

After going through this lesson, you will be able to

- Define what a class is
- Understand the process of object instantiation and memory allocation.
- Understand the concept of instance variables
- Explain the role of access specifiers
- Define constructors and their role in object initialization.

STRUCTURE:

6.1 Classes

6.1.1 Properties of a Java Class

6.1.2 Components of a class

6.1.2.1 Class Declaration

6.1.2.2. Fields (Instance Variables)

6.1.2.3. Constructors

6.1.2.4. Methods

6.1.2 5. Access Modifiers

6.1.2.6. Static Members

6.1.2.7. Inner Classes

6.1.2.8. Blocks

6.1.2.9. Comments

6.2 Objects

6.2.1 Declaring Objects in Java

6.2.2 Initializing a Java object

6.2.2.1 By Reference Variable

6.2.2.2 By method

6.2.2.3 By constructor

6.3 Object creation

6.3.1 Using new Keyword

6.3.2 Using Class.forName(String className) Method

6.3.3 Using clone() Method

6.4 Initializing the instance variables

6.4.1. Default Initialization

6.4.2. Explicit Initialization

6.4.3. Initialization Using Constructors

6.4.4. Initialization Using Instance Initialization Blocks

6.4.5. Initialization Using Methods

6.4.6. Initialization Using Setter Methods

6.5 Access Specifiers

6.5.1 Default Access Modifier

6.5.2 Private Access Modifier

6.5.3. Protected Access Modifier

6.5.4 Public Access Modifier

6.6 Constructors

6.6.1 Characteristics of Constructors

6.6.2 Types of Constructors

6.6.2.1 Default Constructor (No-Argument Constructor)

6.6.2.2 Parameterized Constructor

6.6.3 Constructor Overloading

6.6.4 Calling a Superclass Constructor

6.7 Summary

6.8 Technical Terms

6.9 Self-Assessment Questions

6.10 Further Readings

6.1 CLASSES

Java is a programming language that requires the use of classes at all times and forces the developer to use an object model. In order to represent objects that combine together different pieces of data and methods, classes are used as a prototype. Each creature that possesses a state and behavior is referred to as an object. As a consequence of this, we understand a problem in the world in terms of objects, and we carry out actions by calling the set of methods that are connected with those objects.

The Java programming language's class and object concepts are extremely helpful in the development process as well as in the resolution of complex issues.

Within the Java programming language, a class acts as a template for the creation of objects that have common behaviors and properties. A Car class, which represents individual automobiles, is an example of something that embodies attributes and meaning inside a particular context. Classes are used to preserve common attributes and behaviors, which makes the process of creating and managing objects in programming more practical and efficient.

6.1.1 Properties of a Java Class:

- An object can be created using a Java class, which does not consume any memory and acts as a template for creation.
- Variables of a wide variety of types and approaches are included in it. Data members, methods, constructors, nested classes, and interfaces are all allowable components that can be contained within a class.
- The behavior of objects in your program can be organized and defined with the help of this template, which functions as a template
-

Syntax for creating a class:

```
<<access specifier>> class <<ClassName>>
{
// declaration section for
// methods and attributes
}
```

- A class declaration may have zero or more modifiers.
- The keyword `class` is used to declare a class.

- The <<class name>> is a user-defined name of the class, which should be a valid identifier.
- Each class has a body, which is specified inside a pair of braces ({ ... }).
- The body of a class contains its different components, for example, fields, methods, etc

Example :

```
class Car{
    // declaration of private attributes
    private String modelName;
    private String owner;
    private int regNumber;

    // declaration of public constructor
    public Car(String modelName, String owner, int regNumber){
        this.modelName = modelName;
        this.owner = owner;
        this.regNumber = regNumber;
    }

    // declaration of public methods
    public void startEngine(){
        System.out.println("Engine is starting ....");
    }

    public void accelerate(){
        System.out.println("Car is accelerting ...");
    }

    public void stop(){
        System.out.println("Car is stopping ...");
    }
    // prints car attributes
    public void showCarInformation(){
        System.out.println("The car is owned by: " + this.owner);
        System.out.println("Car Model: " + this.modelName);
        System.out.println("Registration Number: " +
String.valueOf(this.regNumber));
    }
}
```

An example of how classes and objects are implemented in Java may be seen in the code that is displayed above. The Car class contains private attributes hidden from the outside world. In other words, if you call out the modelName attribute outside of the scope of the Car class, you will cause an error to be generated by the compiler.

6.1.2 Components of a class:

6.1.2.1 Class Declaration

This is the header of the class that specifies its name, access level, and any other class modifiers. It begins with the class keyword.

Syntax:

```
public class ClassName {
    // class body
}
```

Example:

```
public class Car {  
    // class body  
}
```

6.1.2.2. Fields (Instance Variables)

Fields, also known as instance variables, represent the properties or state of a class. These are variables declared inside the class but outside any method, constructor, or block. Each object of the class can have different values for these variables.

Syntax:

```
private int numberOfDoors;  
public String color;
```

Example:

```
public class Car {  
    private String model;  
    public int year;  
}
```

6.1.2.3. Constructors

A constructor is a special method that is called when an object of the class is created. It is used to initialize the object. Constructors have the same name as the class and do not have a return type.

Syntax:

```
public ClassName(parameters) {  
    // constructor body  
}
```

Example:

```
public class Car {  
    private String model;  
    public int year;  
  
    // Constructor  
    public Car(String model, int year) {  
        this.model = model;  
        this.year = year;  
    }  
}
```

6.1.2.4. Methods

Methods define the behaviors of the objects created from the class. They are blocks of code that perform a specific task and can be called upon to execute. Methods can have return types and parameters.

Syntax:

```
public returnType methodName(parameters) {  
    // method body  
}
```

Example:

```
public class Car {
    private String model;
    public int year;

    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    // Method
    public void displayInfo() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}
```

6.1.2.5. Access Modifiers

Access modifiers control the visibility of the class, fields, constructors, and methods. Common access modifiers include public, private, protected, and package-private (no explicit modifier).

- public: The class, method, or field is accessible from any other class.
- private: The method or field is accessible only within the class it is declared.
- protected: The method or field is accessible within its own package and by subclasses.
- Package-private (default): The method or field is accessible only within its own package.

Example:

```
public class Car {
    private String model; // private access
    public int year;      // public access
}
```

6.1.2.6. Static Members

Static members (fields or methods) belong to the class itself rather than any particular object instance. They are shared among all instances of the class.

- Static Fields: Shared among all objects of the class.
- Static Methods: Can be called without creating an instance of the class.

Syntax:

```
public static returnType methodName(parameters) {
    // static method body
}
```

Example:

```
public class Car {
    private String model;
    public int year;
    public static int numberOfCars; // Static field

    public static void showNumberOfCars() { // Static method
        System.out.println("Total Cars: " + numberOfCars);
    }
}
```

6.1.2.7. Inner Classes

A class can have another class defined inside it, known as an inner class. Inner classes are useful for logically grouping classes that are only used in one place.

Syntax:

```
public class OuterClass {
    class InnerClass {
        // inner class body
    }
}
```

Example:

```
public class Car {
    private String model;
    public int year;

    // Inner class
    class Engine {
        public void start() {
            System.out.println("Engine started.");
        }
    }
}
```

6.1.2.8. Blocks

Java allows the use of blocks to initialize fields or perform actions that are shared among constructors. These blocks are executed in the order they are defined.

- Instance Initialization Block: Runs every time an instance is created.
- Static Initialization Block: Runs once when the class is loaded.

Example:

```
public class Car {
    static {
        System.out.println("Static block executed");
    }
    {
        System.out.println("Instance block executed");
    }
    public Car() {
        System.out.println("Constructor executed");
    }
}
```

6.1.2.9. Comments

Comments are used to make the code more understandable and are ignored by the compiler. Java supports single-line (`//`) and multi-line (`/* ... */`) comments, as well as Javadoc comments (`/ ... */`) for generating documentation.

Example:

```
public class Car {
    // Single-line comment
    private String model; // Field to store car model

    /
    * Constructor to initialize car object
    */
    public Car(String model) {
        this.model = model;
    }
}
```

6.2 OBJECTS

A real-world entity is modeled by an object in the world. When modeling entities, it is necessary to determine the state of the object as well as the set of actions that may be carried out within that object. Object-oriented programming relies heavily on this method of thinking as its foundation.

- In Java, the root class of all objects that have been instantiated is called an *Object*.
- Instantiated objects are names that refer to an instance of the class.

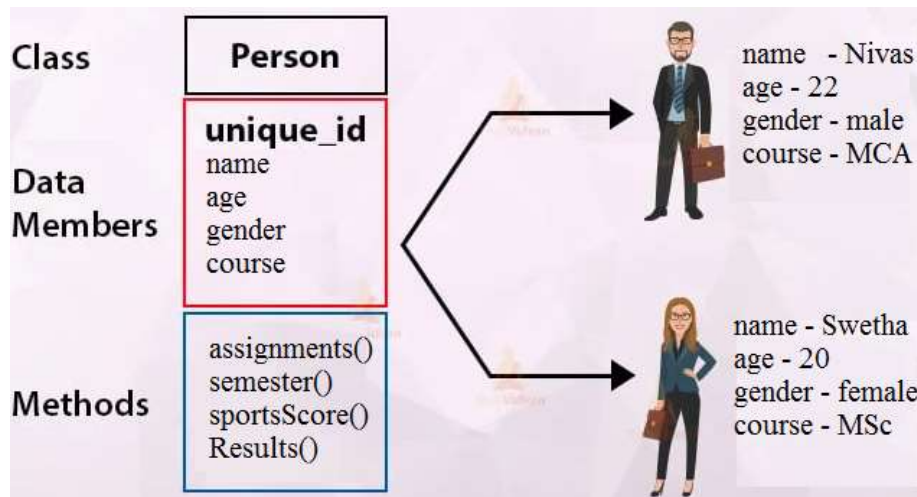


Figure 6.1 class and objects

6.2.1 Declaring Objects in Java

```
Sphere sphere = new Sphere(10);
```

The first two tokens in the code snippet above declare an object in Java. When we read into the code, it means creating a new instance of Sphere and initializing its radius to 10.

6.2.2 Initializing a Java object

As mentioned, a constructor initializes an object in Java. Multiple class constructors mean that instantiating an object can have different signatures. We use the new NameOfClass([arguments_values]) to initialise a new object.

There are mainly three ways to initialize an object in Java:

6.2.2.1 By Reference Variable

A constructor may accept reference type parameters.

Example:

```
class Cube extends AbstractShape{
    private double side;
    public Cube(Cube cubeReference){
        this(this.side);
    }
}
```

The above code effectively copies the content of arguments passed in the constructor of type Cube.

6.2.2.2 By method

An object may be initialized with a setter function that alters the class's internal state and may be accessed with a getter function. This initialization allows only one instance of the class to be modified dynamically: one can alter that of the object at runtime.

Example:

```
class Cube extends AbstractShape{
    private double side;

    public Cube(){
        System.out.println("Cube is instantiated.");
    }
    public void setSide(double side){
        this.side = side;
    }

    public double getSide(){
        return this.side;
    }
    // ...
}

class Main{
    public static void main(String [] args){
        Cube cube = new Cube();
        cube.setSide(5.7);
        System.out.println("Internal state of the cube: " + cube.getSide());
        cube.setSide(11.2);
        System.out.println("Internal state of the cube: " + cube.getSide());
    }
}
```

6.2.2.3 By constructor

Finally, an object may be initialized by specifying arguments passed in a constructor. A constructor is a unique method that initializes the state of class instance.

6.3 OBJECT CREATION

Different Ways to Create Objects in Java

There are different ways to instantiate an object in Java; this section aims to discuss and implement each style.

6.3.1 Using new Keyword

This is the most direct form of object creation in Java. This style tells Java to initialize a class instance and assign a reference to it in a named object, in this case, cube.

```
Cube cube = new Cube(4.5);
```

6.3.2 Using Class.forName(String className) Method

This style can be attained if the class has a public constructor. The Class.forName() method does not yet instantiate an object; to do so, the newInstance() has to be typecast in the given class.

```
class Cube {
    public Cube(){
        System.out.println("Cube is instantiated.");
    }
    // ...
}
```

```
public class Main{
    public static void main(String[] args) throws ClassNotFoundException,
                                                InstantiationException,
                                                IllegalAccessException{
        Class obj = Class.forName("Cube");
        Cube cube = (Cube)obj.newInstance();
    }
}
```

6.3.3. Using clone() Method

It requires the object to implement a Cloneable interface. Since objects are passed by reference in Java, changes that happen somewhere in the program affect the internal state of that object. In cases where we do not want unintended side effects to happen, we can copy the entire contents of an object and treat it independently.

```
class Cube implements Cloneable{
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public Cube(double x){
        System.out.println("instantiated with " + x);
    }
    // ...
}

public class Main{
    public static void main(String[] args) throws
CloneNotSupportedException{
        Cube cube1 = new Cube(4.5);
        Cube cube2 = (Cube)cube1.clone();
    }
}
```

6.4 INITIALIZING THE INSTANCE VARIABLES

An instance variable in Java is a type of variable that is defined within a class but outside any method, constructor, or block. It is also known as a non-static field. Each instance (or object) of a class has its own copy of the instance variable, meaning that each object can have different values for these variables.

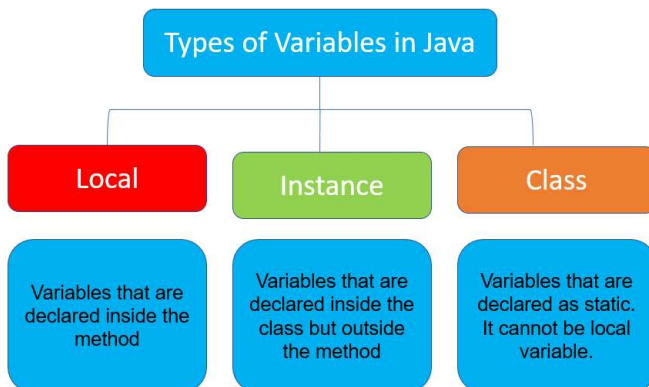


Figure 6.2 types of variables

There are several ways to initialize instance variables in Java:

6.4.1. Default Initialization

Java automatically initializes instance variables to their default values if they are not explicitly initialized.

- Numeric types ('int', 'float', 'double', etc.) are initialized to '0' or '0.0'.
- Characters ('char') are initialized to "\u0000" (the null character).
- Booleans ('boolean') are initialized to 'false'.
- Objects (any reference type, such as 'String') are initialized to 'null'.

Example:

```
public class Car {
    // Instance variables
    private String model;
    private int year;

    public void displayInfo() {
        System.out.println("Model: " + model); // Output: Model: null
        System.out.println("Year: " + year);    // Output: Year: 0
    }

    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.displayInfo();
    }
}
```

6.4.2. Explicit Initialization

Instance variables can be initialized explicitly when they are declared. This method assigns a specific value to the variable when the class instance is created.

Example:

```
public class Car {
    // Explicit initialization
    private String model = "Toyota";
    private int year = 2020;

    public void displayInfo() {
        System.out.println("Model: " + model); // Output: Model: Toyota
        System.out.println("Year: " + year);    // Output: Year: 2020
    }

    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.displayInfo();
    }
}
```

6.4.3. Initialization Using Constructors

Constructors are special methods used to initialize objects. When an object is created, the constructor is called, and you can use it to initialize instance variables.

Example:

```
public class Car {
    private String model;
    private int year;

    // Constructor to initialize instance variables
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public void displayInfo() {
        System.out.println("Model: " + model); // Output: Model: Honda
        System.out.println("Year: " + year);   // Output: Year: 2022
    }

    public static void main(String[] args) {
        Car myCar = new Car("Honda", 2022);
        myCar.displayInfo();
    }
}
```

6.4.4. Initialization Using Instance Initialization Blocks

Instance initialization blocks are blocks of code inside a class that are executed whenever an object of the class is created. They are executed before the constructor.

Example:

```
public class Car {
    private String model;
    private int year;

    // Instance initialization block
    {
        model = "Ford";
        year = 2019;
    }

    public void displayInfo() {
        System.out.println("Model: " + model); // Output: Model: Ford
        System.out.println("Year: " + year);   // Output: Year: 2019
    }

    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.displayInfo();
    }
}
```

6.4.5. Initialization Using Methods

Instance variables can also be initialized using methods. This approach allows more complex logic to be applied when initializing values.

Example:

```
public class Car {
    private String model;
    private int year;

    public void initialize(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public void displayInfo() {
        System.out.println("Model: " + model); // Output: Model: BMW
        System.out.println("Year: " + year);   // Output: Year: 2021
    }

    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.initialize("BMW", 2021);
        myCar.displayInfo();
    }
}
```

6.4.6. Initialization Using Setter Methods

Setter methods are typically used to set or update the values of instance variables after the object has been created. This approach is common in JavaBeans and provides a controlled way to modify the object's state.

Example:

```
public class Car {
    private String model;
    private int year;

    // Setter method for model
    public void setModel(String model) {
        this.model = model;
    }

    // Setter method for year
    public void setYear(int year) {
        this.year = year;
    }

    public void displayInfo() {
        System.out.println("Model: " + model); // Output: Model: Audi
        System.out.println("Year: " + year);   // Output: Year: 2018
    }

    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.setModel("Audi");
        myCar.setYear(2018);
        myCar.displayInfo();
    }
}
```

6.5 ACCESS SPECIFIERS

Access Modifiers in Java impose limitations on the extent of the class, instance variables, methods, and constructor's scope.

In Java, there exist four access modifiers: Default, Private, Protected, and Public. Access modifiers in Java regulate visibility. Keeping personal income information within the family limits access to private information. Public grants unrestricted access, similar to complete awareness of your name by everyone. Protected is analogous to public, but with some restricted scope. The default value functions as a fundamental reference point, located within its package. These modifiers control the visibility of entities like as variables, constructors, and methods.

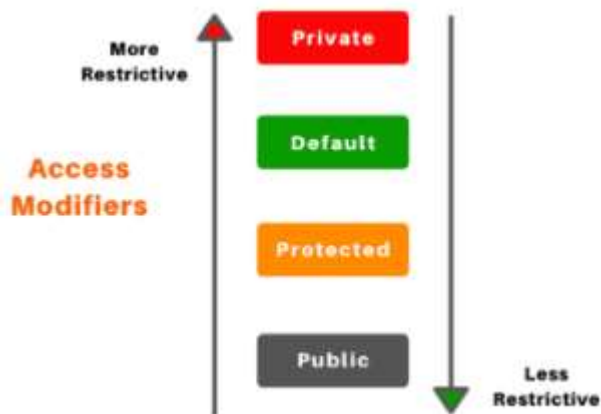


Figure 6.3 types access modifiers

6.5.1 Default Access Modifier

Where no access modifier is expressly provided to class members and methods in Java, they default to package-private access. Consequently, they are exclusively accessible within the same package, hence commonly known as package-private.

Real-world analogy: Envision configuring your Facebook privacy option to "visible only to your friends". Package-private access limits access to members and methods only within the same package, similar to regulating the visibility of your status to just known friends.

Access within the same package: Members and methods with default access can be accessed freely within the same package. Access from another package: Attempting to access default members from another package results in an error, as default access doesn't permit this.

Example 1

In the example below is a package First containing two classes. We are trying to access the first-class default method in the second class.

First Package

```
package First;
public class University {
    int id = 1;
    void print() {
        System.out.println("This is the University class");
    }
}
```

```
class Hello {
    public static void main(String[] args) {
        University ob = new University();
        //This line will call the print() method, which is having the default
        modifier
        ob.print();
    }
}
```

Output:

C:\Users\Surya\.jdk\openjdk-16.0.2\bin\java.exe

This is the University class

The program will run successfully because default members and the methods can be accessed in the same package.

Example 2

In the below example, we are accessing the default method of the First package class in another package, i.e. Second.

Second Package

```
package Second;
import First.University;

public class MyProgs {

    public static void main(String[] args) {
        University ob = new University();

        //This line will cause an error
        ob.print();
    }
}
```

Output:

C:\Users\Surya\Desktop\CP\src\Second\MyProgs.java

java: print() is not public in First. University cannot be accessed from outside package

We got an error because the default access modifiers in Java don't allow us to access the members and methods in another package. We are accessing the default method in another package in the above program.

6.5.2. Private Access Modifier

The private modifier in programming limits access to certain data members and methods within a class. It's like setting your Facebook status to "only me" - only you can see it, and similarly, only the class itself can access private members. Even other classes in the same package can't access them.

Example

In the below class, we have two private instance members and a constructor, as well as a method of private type.

```
class University {

    //Private members
    private int roll;
    private String name;

    //Parameterised Constructor
```

```
University(int a) {
    System.out.print(a);
}

//Default constructor private
private University() {}

//Private method
private void print() {
    System.out.println("This is the University class");
}
}

class Main {

    public static void main(String args[]) {
        //Creating the instance of the University class
        //This will successful run
        University obl = new University(1);

        //Creating another instance of University class
        //This will cause an error
        University ob = new University();

        //These two lines also cause errors.
        ob.name = "Surya";
        ob.print();
    }
}
```

Output:

C:\Users\Surya\Desktop\CP\src\University.java

java: construcor University in class University cannot be applied to given types;

required: no arguments

found: no arguments

reason: University() has private access in University

C:\Users\Surya\Desktop\CP\src\University.java

java: name has private access in University

C:\Users\Surya\Desktop\CP\src\University.java

java: print has private access in University

6.5.3. Protected Access Modifier

In Java, protected access is a valuable feature that enables access both within the same package and by subclasses, regardless of their membership in a separate package. Explicitly accessing protected members from another package requires extending the class that has those members. This entails instantiating a subclass in the alternative language package. Mere creation of an object belonging to the class does not provide access to its protected members; it is necessary to inherit them through a derived class.

There are two packages, First and Second, The First package contains one public class and a protected prefixed method, and we are trying to access this method in another package in two ways.

- Creating an instance of the University class (declared in First package)
- Inheriting the University class in the MyProgs class of the Second package.

First package:

```
package First;
public class University {
    int id = 1;
    protected void print() {
        System.out.println("This is the University class");
    }
}
```

Second package:

```
package Second;
import First.University;
class MyProgs extends University {
    public static void main(String[] args) {
        University ob = new University();
        //This line will cause an error
        ob.print();

        MyProgs ob1 = new MyProgs();
        //This line will not cause an error
        ob1.print();
    }
}
```

The line `ob.print()` will cause an error because we cannot access the protected method outside its package, but we can access it by inheriting it in some other class of different packages, that's why `ob1.print()` will not cause any error.

6.5.4 Public Access Modifier

The public access modifier in Java means there are no restrictions on accessing the methods, classes, or instance members of a particular class. It allows access from any package and any class. A real-life example is setting a Facebook status to "public," allowing anyone on Facebook, whether a friend or not, to see it. This flexibility makes the public modifier useful for wide accessibility.

Example

Consider two packages named First and Second with two public classes.

The University class of the First package contains one instance member and one method, i.e. `print()`. Both are of public type. Let's try to access them in another class of different packages, i.e. in the Second package.

First package:

```
package First;
public class University {

    //Instance member
    public int id = 1;

    //Class method
    public void print() {
        System.out.println("This is the University class");
    }
}
```

Second package:

```
package Second;
import First.University;
public class MyProgs {
    public static void main(String[] args) {
        //Creating the instance of the University class
        University ob = new University();
        //Accessing the instance member of University class
        System.out.println(ob.id); //print 1
        //This is the University class will print
        ob.print();
    }
}
```

Output:

1
This is the University class

We got the correct output, because we can access the public modifier's methods or instance variables in any class of the same or different package.

Access Modifier	Within class	Within package	Outside the package	Outside package by subclass
Private	YES	NO	NO	NO
Default	YES	YES	NO	NO
Protected	YES	YES	NO	YES
Public	YES	YES	YES	YES

Figure 6.4 summary of access modifiers

6.6 CONSTRUCTORS

A constructor in Java is a special type of method that is automatically invoked when an object of a class is created. The primary purpose of a constructor is to initialize the newly created object. It sets initial values for the object's instance variables and performs any other setup or configuration that the object needs.

6.6.1 Characteristics of Constructors

- 1. Same Name as the Class:** A constructor must have the same name as the class in which it resides. This is how the Java compiler identifies it as a constructor rather than a regular method.
- 2. No Return Type:** Constructors do not have a return type, not even 'void'. The lack of a return type distinguishes constructors from normal methods.
- 3. Called Automatically:** Constructors are automatically called when an object of the class is created using the 'new' keyword.

4. Cannot be Called Explicitly: Unlike other methods, constructors cannot be called explicitly using the dot ('.') operator. They are invoked only during object creation.

5. Can Be Overloaded: A class can have multiple constructors with different parameter lists (constructor overloading). This allows objects to be created in different ways with different initializations.

6. No Inheritance: Constructors are not inherited by subclasses. However, a subclass can call a superclass constructor using the 'super' keyword.

6.6.2 Types of Constructors

1. Default Constructor (No-Argument Constructor)
2. Parameterized Constructor

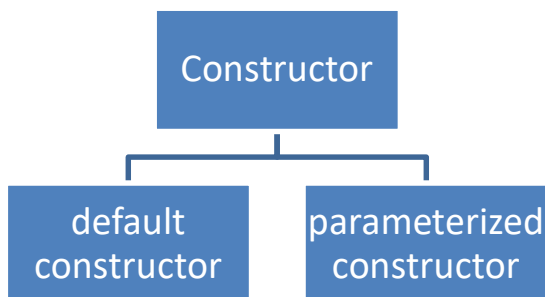


Figure 6.5 types of constructors

6.6.2.1 Default Constructor (No-Argument Constructor)

A default constructor is a constructor that takes no arguments. If a class does not explicitly define any constructor, the Java compiler automatically provides a default constructor. This default constructor initializes the object's instance variables to their default values.

Example:

```
public class Car {
    String model;
    int year;

    // Default constructor
    public Car() {
        model = "Unknown";
        year = 0;
    }

    public void displayInfo() {
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }

    public static void main(String[] args) {
        Car car = new Car(); // Calls the default constructor
        car.displayInfo(); // Output: Model: Unknown, Year: 0
    }
}
```

6.6.2.2 Parameterized Constructor

A parameterized constructor is a constructor that takes one or more parameters. This type of constructor allows you to initialize objects with specific values when they are created.

```
public class Car
    String model;
    int year;

    // Parameterized constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }
    public void displayInfo() {
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }

    public static void main(String[] args) {
        Car car = new Car("Toyota", 2022); // Calls the parameterized
constructor
        car.displayInfo(); // Output: Model: Toyota, Year: 2022
    }
}
```

6.6.3 Constructor Overloading

Constructor overloading in Java means having more than one constructor in a class with different parameter lists. This is useful when you want to provide multiple ways to initialize an object.

Example:

```
public class Car {
    String model;
    int year;

    // Default constructor
    public Car() {
        model = "Unknown";
        year = 0;
    }

    // Parameterized constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public void displayInfo() {
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }

    public static void main(String[] args) {
        Car car1 = new Car(); // Calls the default constructor
        Car car2 = new Car("Honda", 2021); // Calls the parameterized
constructor
        car1.displayInfo(); // Output: Model: Unknown, Year: 0
        car2.displayInfo(); // Output: Model: Honda, Year: 2021
    }
}
```

6.6.4 Calling a Superclass Constructor

In a subclass, you can call a constructor of its superclass using the 'super' keyword. This is typically done when you want to extend the initialization process defined in the superclass.

Example:

```
public class Vehicle {
    String type;

    // Parameterized constructor
    public Vehicle(String type) {
        this.type = type;
    }
}

public class Car extends Vehicle {
    String model;

    // Parameterized constructor
    public Car(String type, String model) {
        super(type); // Calls the constructor of Vehicle class
        this.model = model;
    }

    public void displayInfo() {
        System.out.println("Type: " + type);
        System.out.println("Model: " + model);
    }

    public static void main(String[] args) {
        Car car = new Car("Sedan", "Toyota");
        car.displayInfo(); // Output: Type: Sedan, Model: Toyota
    }
}
```

6.7 SUMMARY

The chapter "Classes and Objects in Java" introduces the foundational concepts of object oriented programming in Java. It begins by explaining classes as blueprints for creating objects, which are instances of these classes, embodying the real-world entities. The chapter covers object creation, detailing how to instantiate objects using the new keyword and discusses various ways to initialize instance variables, which hold the state of an object.

It explores access specifiers (public, protected, private, and default) that control the visibility and accessibility of class members, ensuring encapsulation and security. Lastly, the chapter delves into constructors, special methods used to initialize new objects, including both default and parameterized constructors, demonstrating how to provide initial values and set up necessary conditions for newly created objects

6.8 TECHNICAL TERMS

class, object, instance variable, private, public, protected, constructor

6.9 SELF ASSESSMENT QUESTIONS

Essay questions:

1. What is a class in Java, and how does it relate to objects?
2. How do you create an object in Java? Provide an example.
3. What are instance variables, and how are they different from local variables?
4. Explain the purpose of the private access specifier in Java.
5. What is the difference between a default constructor and a parameterized constructor?

Short Answer Questions:

1. Describe the relationship between classes and objects in Java. Provide examples to illustrate your explanation.
2. Discuss the importance of constructors in Java. How do constructors enhance object creation and initialization? Provide examples of different types of constructors and how they can be used in a Java program.
3. Explain the concept of access specifiers in Java and how they contribute to encapsulation and data hiding with examples
4. How do instance variables get initialized in Java? Discuss the different ways of initializing instance variables and the impact of each method on object behavior.

6.10 SUGGESTED READINGS

- 1) Herbert Schildt and Dale Skrien “Java Fundamentals –A comprehensive Introduction”, McGraw Hill, 1st Edition, 2013.
- 2) Herbert Schildt, “Java the complete reference”, McGraw Hill, Osborne, 11th Edition, 2018.
- 3) T. Budd “Understanding Object-Oriented Programming with Java”, Pearson Education, Updated Edition (New Java 2 Coverage), 1999 REFERENCE BOOKS:
- 4) P.J. Dietel and H.M. Dietel “Java How to program”, Prentice Hall, 6th Edition, 2005.
- 5) P. Radha Krishna “Object Oriented programming through Java”, CRC Press, 1st Edition, 2007.
- 6) Malhotra and S. Choudhary “Programming in Java”, Oxford University Press, 2nd Edition, 2014

Dr. U. SURYA KAMESWARI

LESSON- 7

INHERITANCE

OBJECTIVES

By the end of this chapter, you should be able to:

- Understanding the Concept of Inheritance
- Exploring Different Types of Inheritance
- Applying Inheritance to Real-World Problems
- Recognizing the Limitations of Inheritance
- Define Key Inheritance Terminology
- Create Simple Inheritance Hierarchies
- Use the super Keyword Effectively
- Apply Inheritance in Object-Oriented Design

STRUCTURE

- 7.1 Introduction
- 7.2 Basic Syntax and Terminology
- 7.3 Types of Inheritance
- 7.4 Method Overriding
- 7.5 Best Practices
- 7.6 Case Study: Inheritance in a Real-World Application
- 7.7 Summary
- 7.8 Technical Terms
- 7.9 Self-Assessment Questions
- 7.10 Suggested Readings

7.1 INTRODUCTION

Inheritance is a fundamental concept in object-oriented programming that allows a new class to acquire the properties and behaviors of an existing class. By enabling one class (the subclass) to inherit fields and methods from another class (the superclass), inheritance promotes code reuse, simplifies code maintenance, and establishes a natural hierarchy among classes. This concept not only reduces redundancy but also facilitates the creation of more flexible and scalable programs, as it allows developers to build on existing code without modifying it directly.

❖ Importance of Inheritance

Inheritance is a core concept in object-oriented programming (OOP) that allows a new class to derive properties and behaviors from an existing class. In Java, inheritance enables one class, known as a subclass or child class, to inherit fields and methods from another class, referred to as a superclass or parent class. This mechanism not only promotes code reuse by allowing new classes to build upon existing ones but also establishes a hierarchical relationship between classes, reflecting real-world structures and relationships. Through inheritance, developers can create more modular, maintainable, and scalable software, as common functionality is centralized in super classes and extended or customized in subclasses.

Inheritance is crucial in Java and object-oriented programming for several reasons:

- **Code Reusability:** Inheritance allows developers to reuse existing code by creating new classes based on existing ones. This reduces redundancy and the effort required to write new code, as common functionality can be inherited from a superclass.
- **Logical Class Hierarchy:** Inheritance helps organize code into a logical hierarchy, reflecting real-world relationships. For example, a "Vehicle" class can serve as a superclass for "Car," "Bike," and "Truck" subclasses, each inheriting common properties while introducing specific features.
- **Simplified Maintenance:** When a change is made in the superclass, it automatically propagates to all subclasses, making code easier to maintain and update. This reduces the likelihood of errors and inconsistencies across the codebase.
- **Polymorphism:** Inheritance enables polymorphism, where a subclass can be treated as an instance of its superclass. This allows for more flexible and dynamic code, as methods can be overridden in subclasses to provide specific implementations while maintaining a common interface.
- **Extensibility:** Inheritance makes it easier to extend existing code. Developers can add new features or modify behaviors in subclasses without altering the original superclass code, ensuring that enhancements are made without disrupting existing functionality.
- **Encapsulation and Abstraction:** Inheritance supports encapsulation by allowing a subclass to access protected and public members of the superclass while hiding its internal implementation details. It also aids in abstraction by allowing higher-level classes to represent general concepts, while subclasses provide concrete implementations.

Overall, inheritance is a powerful tool that promotes efficient, organized, and scalable software development, making it an essential concept in Java and object-oriented programming.

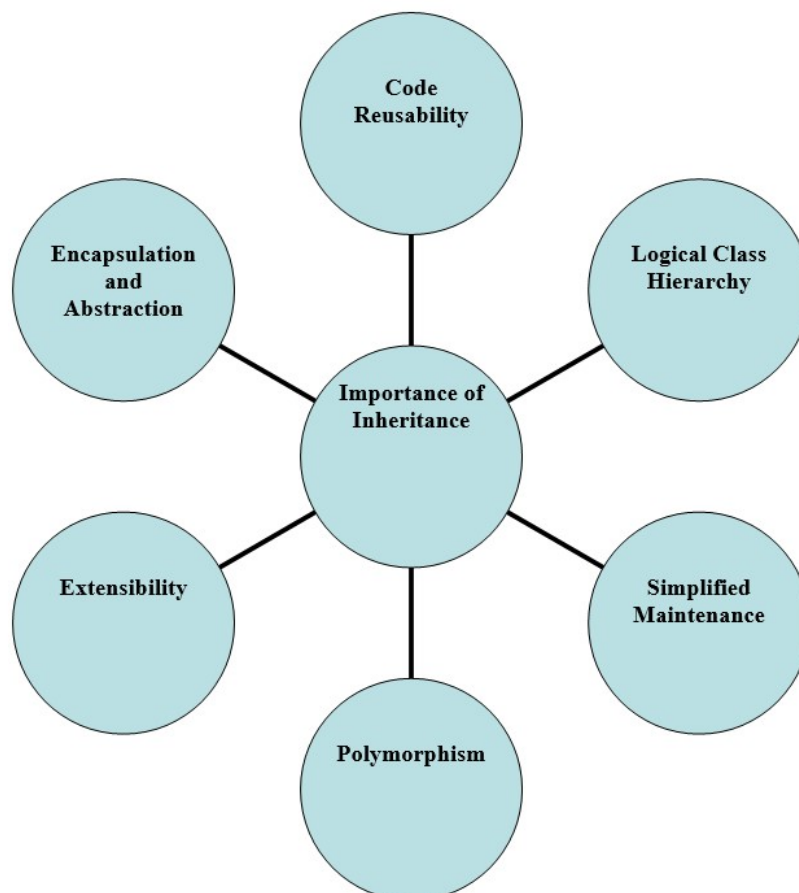


Fig 7.1 Importance of Inheritance

7.2 BASIC SYNTAX AND TERMINOLOGY

In Java, inheritance is implemented using the `extends` keyword, which allows a class to inherit properties and behaviors from another class. Understanding the basic syntax and terminology is essential for effectively utilizing inheritance in your programs.

❖ Superclass and Subclass

- **Superclass (Parent Class):** The class from which properties and methods are inherited. It represents a more general concept in the hierarchy.
 - **Example:** In a class hierarchy where `Vehicle` is a superclass, it might define common attributes like `speed` and methods like `move()`.
- **Subclass (Child Class):** The class that inherits from the superclass. It represents a more specific concept and can add new properties or override existing ones from the superclass.
 - **Example:** `Car` and `Bike` might be subclasses of `Vehicle`, inheriting `speed` and `move()` while adding specific attributes like `numDoors` for `Car`.

❖ The `extends` Keyword

- The `extends` keyword is used in the class declaration to signify that a class is inheriting from another class.

- **Syntax:**

```
class SubclassName extends SuperclassName {  
    // Additional fields and methods  
}
```

- **Example:**

```
class Vehicle {  
    int speed;  
    void move() {  
        System.out.println("Vehicle is moving");  
    }  
}  
  
class Car extends Vehicle {  
    int numDoors;  
    void display() {  
        System.out.println("Car has " + numDoors + " doors and moves at speed " + speed);  
    }  
}
```

❖ The `super` Keyword

- The `super` keyword is used in a subclass to refer to the superclass. It can be used to:

- Call a superclass constructor:

```
class Car extends Vehicle {  
    Car() {  
        super(); // Calls the constructor of Vehicle  
    }  
}
```

- Access a superclass method:

```
class Car extends Vehicle {  
    @Override  
    void move() {  
        super.move(); // Calls the move() method of Vehicle  
    }  
}
```

```
    System.out.println("Car is moving");  
  }  
}
```

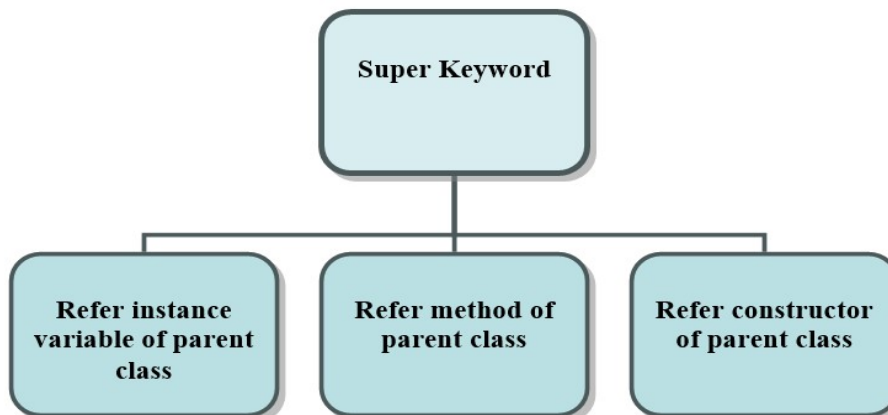


Fig 7.2 Super Keyword in Java

❖ Access Modifiers and Inheritance

- **Public:** Public members of a superclass are accessible in the subclass.
- **Protected:** Protected members are accessible in the subclass and within the same package.
- **Private:** Private members are not accessible in the subclass. However, access can be provided through public or protected getter and setter methods.
- **Default (Package-Private):** Members with no explicit access modifier (default) are accessible within the same package but not in subclasses outside the package.

7.3 TYPES OF INHERITANCE

In Java, inheritance allows a class to inherit properties and behaviors from another class. There are several types of inheritance, each defining a different way in which classes can relate to each other. However, Java does not support multiple inheritance (where a class inherits from more than one class) due to the complexity and ambiguity it can introduce. Below are the primary types of inheritance in Java shown in Figure 8.3 and details explained in below:

- ❖ **Single Inheritance:** One class inherits from one superclass.
- ❖ **Multilevel Inheritance:** A class inherits from a derived class, forming a chain of inheritance.
- ❖ **Hierarchical Inheritance:** Multiple classes inherit from the same superclass.
- ❖ **Multiple Inheritance (Through Interfaces):** A class implements multiple interfaces, allowing for multiple inheritance.
- ❖ **Hybrid Inheritance:** A combination of different types of inheritance, typically implemented using interfaces.

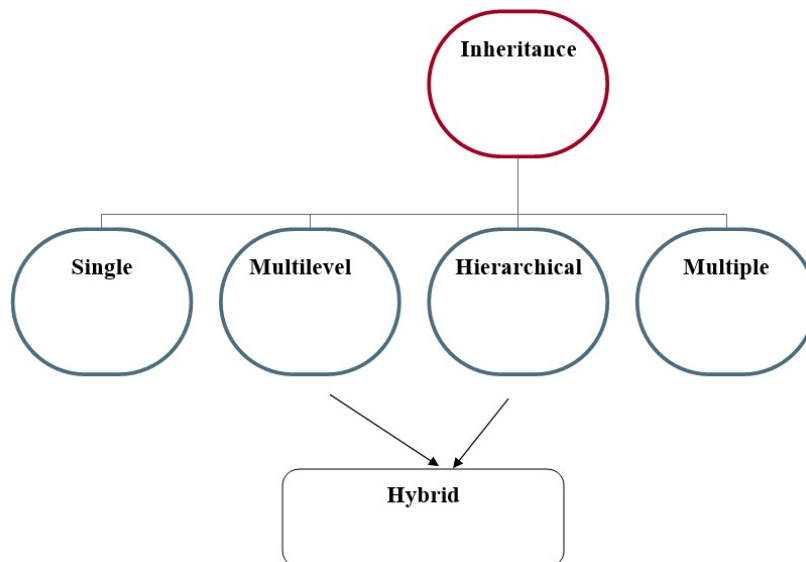


Fig 7.3 Types of Inheritance

❖ **Single Inheritance**

Single inheritance is when a class inherits from only one superclass. This is the most common type of inheritance in Java.

- Example:

```
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
```

```
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}
```

In this example, the Dog class inherits from the Animal class. Dog can use the eat() method from Animal in addition to its own bark() method.

❖ **Multilevel Inheritance**

Multilevel inheritance occurs when a class is derived from another derived class, creating a chain of inheritance.

- Example:

```
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
```

```
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}
```

```
class Puppy extends Dog {
    void weep() {
        System.out.println("Puppy is weeping");
    }
}
```

Here, the Puppy class inherits from Dog, which in turn inherits from Animal. Puppy can use the eat() method from Animal and the bark() method from Dog, as well as its own weep() method.

❖ Hierarchical Inheritance

Hierarchical inheritance occurs when multiple classes inherit from the same superclass.

- Example:

```
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}
class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing");
    }
}
```

In this example, both Dog and Cat classes inherit from the Animal class. Each subclass has its own specific methods (bark() and meow()) in addition to the inherited eat() method.

❖ Multiple Inheritance

Java does not support multiple inheritance through classes due to the "diamond problem" (ambiguity caused when a class inherits from two classes that have a method with the same signature). However, Java allows multiple inheritance through interfaces.

- Example:

```
interface Animal {
    void eat();
}
interface Pet {
    void play();
}
class Dog implements Animal, Pet {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }
    @Override
    public void play() {
        System.out.println("Dog is playing");
    }
}
```

Here, the Dog class implements two interfaces, Animal and Pet, effectively achieving multiple inheritance.

❖ **Hybrid Inheritance**

Hybrid inheritance is a combination of two or more types of inheritance. In Java, hybrid inheritance is not supported directly because it often involves multiple inheritance, which can lead to ambiguity. However, it can be achieved using interfaces.

• **Example:**

```
interface Animal {
    void eat();
}
class Mammal {
    void sleep() {
        System.out.println("Mammal is sleeping");
    }
}
class Dog extends Mammal implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }
}
```

In this example, Dog class inherits from Mammal (single inheritance) and implements Animal interface (multiple inheritance through interfaces), creating a hybrid inheritance scenario.

Table 7.1 Differences between various types of inheritance

Type	Description
• Single Inheritance	• a derived class is created from a single base class.
• Multi-level Inheritance	• a derived class is created from another derived class.
• Multiple Inheritance	• a derived class is created from more than one base class
• Hierarchical Inheritance	• more than one derived class is created from a single base class
• Hybrid Inheritance	• a combination of more than one inheritance

7.4 METHOD OVERLOADING

Method overloading occurs when two or more methods in the same class have the same name but different parameter lists (different in number, type, or order of parameters). The compiler determines which method to call based on the method signature at compile time.

Example of Method Overloading:

```
class MathOperation {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }
    // Overloaded method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }
    // Overloaded method to add two double values
    double add(double a, double b) {
        return a + b;
    }
}

public class TestOverloading {
    public static void main(String[] args) {
        MathOperation mo = new MathOperation();
        // Calling the method with two integers
        System.out.println("Sum of two integers: " + mo.add(10, 20));
        // Calling the method with three integers
        System.out.println("Sum of three integers: " + mo.add(10, 20, 30));
        // Calling the method with two double values
        System.out.println("Sum of two doubles: " + mo.add(10.5, 20.5));
    }
}
```

Output:

Sum of two integers: 30

Sum of three integers: 60

Sum of two doubles: 31.0

In this example, the add method is overloaded to handle different types of input. The correct method is selected at compile time based on the arguments passed.

7.5 BEST PRACTICES

Inheritance is a powerful feature in Java that allows a class to inherit properties and behaviors from another class, promoting code reuse and organization. However, it can also lead to complexities and potential pitfalls if not used carefully. Here are some best practices to follow when using inheritance in Java and described in Fig 7.4:



Fig 7.4 Best Practices of Inheritance

❖ Favor Composition Over Inheritance

- Composition involves building classes by including instances of other classes that implement the desired functionality. This is often more flexible and less error-prone than inheritance.
- Why? Inheritance tightly couples the parent and child classes, making it harder to change one without affecting the other. Composition allows for more modular, maintainable, and reusable code.

Example:

```
class Engine {  
    void start() {  
        System.out.println("Engine started");  
    }  
}
```

```
class Car {  
    private Engine engine;  
  
    Car() {  
        engine = new Engine();  
    }  
  
    void start() {  
        engine.start();  
        System.out.println("Car started");  
    }  
}
```

Instead of inheriting from an Engine class, the Car class uses composition to include an Engine instance.

❖ Use Inheritance for "Is-A" Relationships

- Ensure that the relationship between the superclass and subclass genuinely reflects an "is-a" relationship. The subclass should be a more specific version of the superclass.
- Why? Misusing inheritance can lead to improper design where subclasses inherit methods or properties that do not logically apply to them, leading to confusion and maintenance difficulties.
- **Example:**
 - A Dog class should inherit from an Animal class because a dog "is an" animal.
 - Avoid scenarios like Square inheriting from Rectangle if their relationship isn't truly "is-a."

❖ Keep Class Hierarchies Shallow

- Avoid deep inheritance hierarchies (i.e., many levels of inheritance). Prefer flatter class structures where possible.
- Why? Deep hierarchies can lead to increased complexity, making the code harder to understand, maintain, and debug. Shallow hierarchies are easier to manage.
- **Example:**
 - If you find yourself creating multiple levels of inheritance, consider whether some of the intermediate classes can be merged or if composition can replace some inheritance.

❖ Avoid Overriding Methods Unnecessarily

- Only override methods when there is a clear need to change or extend the behavior of the superclass method.
- Why? Unnecessary overriding can introduce bugs and make the code harder to follow. If the superclass method behavior is sufficient, there's no need to override it.
- **Example:**
 - If a Vehicle class has a startEngine() method that works for all vehicles, subclasses like Car or Bike should only override it if they need specific behavior.

❖ Use the super Keyword Carefully

- Use the super keyword to access methods and constructors of the superclass, but do so judiciously.
- Why? Misusing super can lead to unexpected behavior, especially if the superclass method is not intended to be extended in a particular way.
- **Example:**

```
class Animal {  
    void sound() {  
        System.out.println("Animal sound");  
    }  
}
```

```
class Dog extends Animal {  
    @Override  
    void sound() {  
        super.sound(); // Calls the superclass method  
        System.out.println("Dog barks");  
    }  
}
```

❖ **Mark Methods and Classes as final When Necessary**

- Use the final keyword to prevent classes from being extended or methods from being overridden when it's not appropriate for them to be modified.
- Why? Preventing further extension of a class or method helps to maintain the integrity of your design and ensures that certain behaviors are not unintentionally altered.
- **Example:**

```
public final class MathUtils {  
    public static final double PI = 3.14159;  
}
```

❖ **Ensure Proper Use of Constructors**

- Ensure that subclass constructors call the appropriate superclass constructor, especially when the superclass does not have a default constructor.
- Why? Failing to properly initialize a superclass can lead to incomplete object creation and potential runtime errors.
- **Example:**

```
class Animal {  
    String name;  
    Animal(String name) {  
        this.name = name;  
    }  
}  
class Dog extends Animal {  
    Dog(String name) {  
        super(name); // Call to superclass constructor  
    }  
}
```

❖ **Be Cautious with Protected Members**

- Use the protected access modifier carefully, as it allows subclasses to access superclass members directly.
 - Why? Overusing protected can expose internal implementation details that should remain encapsulated, leading to tight coupling and potential misuse.
 - Example:
- Prefer private members with appropriate getter/setter methods over protected members to maintain encapsulation.

❖ **Use Abstract Classes for Common Functionality**

- Use abstract classes when you want to define common behavior that multiple subclasses should share while allowing for specific implementations in each subclass.
- Why? Abstract classes provide a way to enforce certain methods while still allowing for flexibility in subclass behavior.
- **Example:**

```
abstract class Animal {  
    abstract void sound();  
  
    void breathe() {  
        System.out.println("Animal is breathing");  
    }  
}  
class Dog extends Animal {
```

```
@Override
void sound() {
    System.out.println("Dog barks");
}
}
```

When using inheritance in Java, it is essential to follow best practices to ensure that your code remains maintainable, flexible, and understandable. By favoring composition over inheritance, keeping hierarchies shallow, and carefully managing method overrides and access modifiers, you can avoid common pitfalls and make the most of inheritance's benefits. Proper use of abstract classes, constructors, and documentation further enhances the effectiveness and clarity of your inheritance-based designs.

7.6 CASE STUDY: INHERITANCE IN A REAL-WORLD APPLICATION

Suppose you are developing a Vehicle Management System for a car rental company. The system needs to manage different types of vehicles, such as cars, trucks, and motorcycles. These vehicles share some common characteristics but also have specific features and behaviors unique to each type. Using inheritance in this scenario allows you to model the commonalities and differences efficiently.

```
class Vehicle {
    private String make;
    private String model;
    private int year;
    private String color;
    private String registrationNumber;

    public Vehicle(String make, String model, int year, String color, String
registrationNumber) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.color = color;
        this.registrationNumber = registrationNumber;
    }

    public void displayDetails() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
        System.out.println("Color: " + color);
        System.out.println("Registration Number: " + registrationNumber);
    }

    public double calculateRentalPrice() {
        return 50.0; // Base rental price for any vehicle
    }
}
```



```
class Car extends Vehicle {
    private int numberOfDoors;
    private boolean isConvertible;

    public Car(String make, String model, int year, String color, String registrationNumber, int
numberOfDoors, boolean isConvertible) {
        super(make, model, year, color, registrationNumber);
        this.numberOfDoors = numberOfDoors;
        this.isConvertible = isConvertible;
    }

    @Override
    public void displayDetails() {
        super.displayDetails();
        System.out.println("Number of Doors: " + numberOfDoors);
        System.out.println("Convertible: " + (isConvertible ? "Yes" : "No"));
    }

    @Override
    public double calculateRentalPrice() {
        double basePrice = super.calculateRentalPrice();
        return isConvertible ? basePrice + 30 : basePrice + 20;
    }
}

class Truck extends Vehicle {
    private double cargoCapacity;
    private boolean hasTrailer;

    public Truck(String make, String model, int year, String color, String registrationNumber,
double cargoCapacity, boolean hasTrailer) {
        super(make, model, year, color, registrationNumber);
        this.cargoCapacity = cargoCapacity;
        this.hasTrailer = hasTrailer;
    }

    @Override
    public void displayDetails() {
        super.displayDetails();
        System.out.println("Cargo Capacity: " + cargoCapacity + " tons");
        System.out.println("Has Trailer: " + (hasTrailer ? "Yes" : "No"));
    }

    @Override
    public double calculateRentalPrice() {
        double basePrice = super.calculateRentalPrice();
        return basePrice + (hasTrailer ? 50 : 40);
    }
}

class Motorcycle extends Vehicle {
    private int engineCapacity;
```

```
public Motorcycle(String make, String model, int year, String color, String
registrationNumber, int engineCapacity) {
    super(make, model, year, color, registrationNumber);
    this.engineCapacity = engineCapacity;
}

@Override
public void displayDetails() {
    super.displayDetails();
    System.out.println("Engine Capacity: " + engineCapacity + " cc");
}

@Override
public double calculateRentalPrice() {
    double basePrice = super.calculateRentalPrice();
    return basePrice + (engineCapacity > 1000 ? 25 : 15);
}
}

public class VehicleManagementSystem {
    public static void main(String[] args) {
        Vehicle car = new Car("Toyota", "Camry", 2020, "Red", "ABC123", 4, false);
        Vehicle truck = new Truck("Ford", "F-150", 2019, "Blue", "XYZ789", 5.0, true);
        Vehicle motorcycle = new Motorcycle("Harley-Davidson", "Sportster", 2021, "Black",
"MNO456", 1200);

        System.out.println("Car Details:");
        car.displayDetails();
        System.out.println("Rental Price: $" + car.calculateRentalPrice());

        System.out.println("\nTruck Details:");
        truck.displayDetails();
        System.out.println("Rental Price: $" + truck.calculateRentalPrice());

        System.out.println("\nMotorcycle Details:");
        motorcycle.displayDetails();
        System.out.println("Rental Price: $" + motorcycle.calculateRentalPrice());
    }
}
```

OUTPUT:

```
Car Details:
Make: Toyota
Model: Camry
Year: 2020
Color: Red
Registration Number: ABC123
Number of Doors: 4
Convertible: No
Rental Price: $70.0
```

Truck Details:

Make: Ford
Model: F-150
Year: 2019
Color: Blue
Registration Number: XYZ789
Cargo Capacity: 5.0 tons
Has Trailer: Yes
Rental Price: \$100.0

Motorcycle Details:

Make: Harley-Davidson
Model: Sportster
Year: 2021
Color: Black
Registration Number: MNO456
Engine Capacity: 1200 cc
Rental Price: \$75.0

This case study illustrates how inheritance in Java can be effectively used to model real-world scenarios. By using inheritance, you can create a flexible, reusable, and maintainable codebase that is easy to extend and adapt to new requirements. The Vehicle Management System demonstrates how different vehicle types share common features through a base class while allowing specific behaviors through subclassing. This approach reduces code duplication, enhances clarity, and provides a strong foundation for future development.

7.7 SUMMARY

Inheritance in Java is a core concept of object-oriented programming that allows one class (the subclass) to inherit fields and methods from another class (the superclass). This enables code reuse and the creation of a hierarchical class structure, where subclasses can extend or modify the behaviors of the superclass. Inheritance supports the "is-a" relationship, ensuring that subclasses can be used interchangeably with their superclass, promoting flexibility and maintainability in code design. It also allows for method overriding, enabling polymorphism, where the same method can have different behaviors in different classes.

7.8 TECHNICAL TERMS

1. Super
2. extends
3. single level
4. Hybrid
5. Multiple
6. Method Overriding

7.9 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain the concept of inheritance in Java and how it promotes code reuse and organization. Provide examples to illustrate your explanation.
2. Discuss the differences between method overloading and method overriding in the context of inheritance. How does Java handle these concepts at compile-time and runtime?
3. Describe how the super keyword is used in Java to access superclass methods and constructors. Provide examples showing its role in constructor chaining and method invocation.
4. What are the benefits and potential pitfalls of using inheritance in Java? Discuss the concept of favoring composition over inheritance and provide examples to support your argument.
5. Explain how the final keyword can be used to prevent inheritance and method overriding in Java. What are the implications of marking a class or method as final?

Short questions:

1. What is inheritance in Java?
2. How does the extends keyword work in Java?
3. What is the difference between method overloading and method overriding?
4. What is the purpose of the super keyword in Java?
5. How does the final keyword affect inheritance?

7.10 SUGGESTED READINGS

1. "Java: The Complete Reference" by Herbert Schildt, 12th Edition (2021), McGraw-Hill Education
2. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition (2005), O'Reilly Media
3. "Effective Java" by Joshua Bloch, 3rd Edition (2018), Addison-Wesley Professional

Dr. KAMPA LAVANYA

LESSON- 8

POLYMORPHISM

OBJECTIVES

By the end of this chapter, you should be able to:

- execute a block of code multiple times, reducing redundancy and ensuring that repetitive tasks are automated.
- systematically access each element in a collection or array, enabling operations such as processing, searching, or modifying data.
- dynamically control the flow of execution based on conditions, allowing for flexible and adaptive programming.
- handle large datasets or perform calculations repeatedly without manually duplicating code.
- manage and update counters, such as for indexing elements, tracking iterations, or controlling loops.

These objectives highlight the importance of loop statements in Java for creating efficient, readable, and maintainable code.

STRUCTURE

- 8.1 Introduction
- 8.2 Importance of Polymorphism In Java
- 8.3 Compile time Polymorphism
- 8.4 Runtime Polymorphism
- 8.5 Polymorphism with Interface
- 8.6 Polymorphism and Abstract Classes
- 8.7 Advantages and Disadvantages of Polymorphism
 - 8.7.1 Advantages
 - 8.7.2 Disadvantages
- 8.8 Common Mistakes and Best Practices
 - 8.8.1 Common Mistakes
 - 8.8.2 Best Practices
- 8.9 Summary
- 8.10 Technical Terms
- 8.11 Self-Assessment Questions
- 8.12 Suggested Readings

8.1 INTRODUCTION

Polymorphism, a core concept in object-oriented programming, refers to the ability of a single function, method, or operator to operate in different ways based on the context. In Java, polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling the same method to behave differently depending on the object it is acting upon. This feature enhances flexibility and reusability in code, making it easier to extend and maintain. Through polymorphism, Java developers can write more generic and scalable programs, where specific behaviors can be altered dynamically at runtime without altering the code that invokes these behaviors.

8.2 IMPORTANCE OF POLYMORPHISM IN JAVA

Polymorphism is a cornerstone of object-oriented programming (OOP) in Java, playing a crucial role in the language's design and usage. Here are the key reasons why polymorphism is important in Java and are shown in Figure 8.1:

- ❖ **Code Reusability:** Polymorphism allows developers to use a single interface to represent different types of objects. This promotes code reusability as the same code can operate on objects of different classes without modification, reducing redundancy and simplifying maintenance.
- ❖ **Flexibility and Extensibility:** Polymorphism enables code to be more flexible and extensible. By programming to interfaces or base classes, new subclasses or implementations can be introduced with minimal changes to existing code. This flexibility allows for easier updates and scaling of applications.
- ❖ **Dynamic Method Dispatch:** With polymorphism, Java supports dynamic method dispatch, where the method to be executed is determined at runtime. This allows for more dynamic and responsive applications, where behavior can be altered based on the actual object type that the reference variable points to at runtime.
- ❖ **Simplified Code Management:** Polymorphism simplifies code management by reducing the complexity associated with conditional statements or type checks. Instead of writing multiple conditional branches to handle different types, a single method call can be used, and polymorphism ensures the correct method is executed based on the object's type.
- ❖ **Enhanced Maintainability:** Since polymorphism leads to a more modular code structure, it enhances the maintainability of the software. Changes in one part of the system (e.g., introducing a new subclass) can be isolated from other parts, leading to fewer bugs and easier testing and debugging.
- ❖ **Design Patterns and Frameworks:** Polymorphism is a foundational concept behind many design patterns and frameworks in Java, such as the Strategy Pattern, Observer Pattern, and Dependency Injection. These patterns leverage polymorphism to create flexible and reusable code structures, essential for building robust enterprise-level applications.



Fig 8.1 Importance of Polymorphism with various factors

Example:

```
interface Animal {
    void sound();
}
class Dog implements Animal {
    public void sound() {
        System.out.println("Woof");
    }
}

class Cat implements Animal {
    public void sound() {
        System.out.println("Meow");
    }
}
```

8.3 COMPILE-TIME POLYMORPHISM

Polymorphism can be classified into two types and described in Figure 8.2.

1. Compile-Time Polymorphism
2. Run-Time Polymorphism

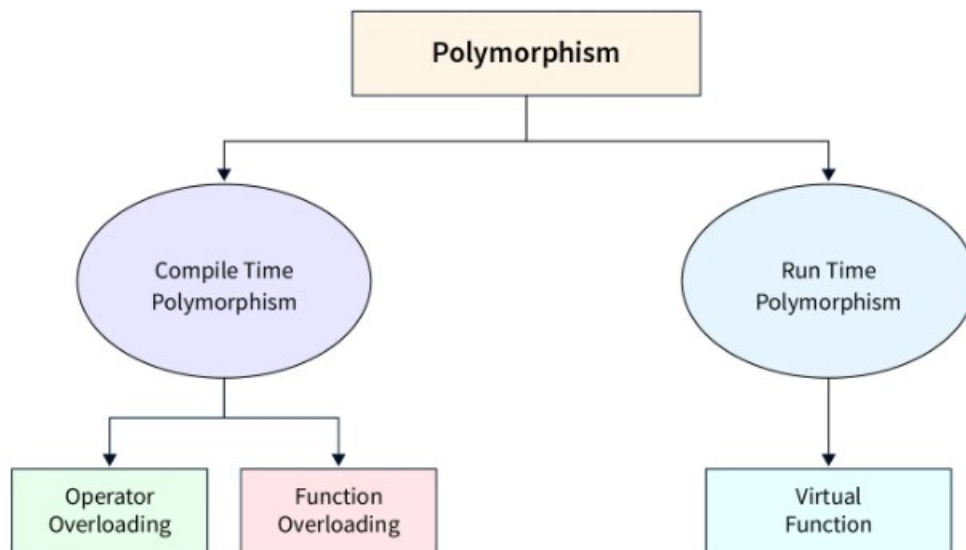


Fig 8.2 Polymorphism classification in Java

Compile-time polymorphism, also known as static polymorphism, is a type of polymorphism that is resolved during the compilation of the program. In Java, compile-time polymorphism is achieved through method overloading and operator overloading (although Java does not support user-defined operator overloading). This form of polymorphism allows a single method or operator to behave differently based on the parameters or context in which it is used.

- **Method Overloading**

Method overloading occurs when two or more methods in the same class have the same name but different parameter lists (different in number, type, or order of parameters). The compiler determines which method to call based on the method signature at compile time.

Example of Method Overloading:

```
class MathOperation {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }
    // Overloaded method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }
    // Overloaded method to add two double values
    double add(double a, double b) {
        return a + b;
    }
}
public class TestOverloading {
    public static void main(String[] args) {
        MathOperation mo = new MathOperation();
        // Calling the method with two integers
        System.out.println("Sum of two integers: " + mo.add(10, 20));
    }
}
```



```
        // Calling the method with three integers
        System.out.println("Sum of three integers: " + mo.add(10, 20, 30));
        // Calling the method with two double values
        System.out.println("Sum of two doubles: " + mo.add(10.5, 20.5));
    }
}
```

Output:

Sum of two integers: 30

Sum of three integers: 60

Sum of two doubles: 31.0

In this example, the add method is overloaded to handle different types of input. The correct method is selected at compile time based on the arguments passed.

Advantages of Compile-time Polymorphism:

1. **Improved Code Readability:** By using the same method name for different types of operations, code becomes more intuitive and easier to understand.
2. **Enhanced Performance:** Since the method to be called is determined at compile time, there is no overhead associated with dynamic method dispatch.
3. **Simplified Maintenance:** Overloading allows related operations to be grouped together under a single method name, making it easier to maintain and update the code.

Limitations of Compile-time Polymorphism:

1. **Limited Flexibility:** Since the method selection is done at compile time, it lacks the flexibility of runtime polymorphism, where decisions can be made dynamically based on actual object types.
2. **No User-defined Operator Overloading:** Java does not allow user-defined operator overloading, unlike some other languages like C++, which limits the scope of compile-time polymorphism.

Compile-time polymorphism in Java is a powerful tool for creating methods that can handle different data types or numbers of parameters while maintaining a clean and readable codebase. It is resolved during the compilation process, which enhances performance but offers less flexibility compared to runtime polymorphism. Understanding and effectively using method overloading can greatly improve the design and functionality of Java programs.

8.4 RUNTIME POLYMORPHISM

Runtime polymorphism, also known as dynamic polymorphism, is a type of polymorphism that is resolved during the execution of a program. In Java, runtime polymorphism is achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass. The decision of which method to invoke is made at runtime, based on the actual object being referred to by the reference variable.

- **Method Overriding**

Method overriding occurs when a subclass has a method with the same name, return type, and parameters as a method in its superclass. The overriding method in the subclass provides a specific implementation that is different from the one in the superclass.

Key Points of Method Overriding:

- The method in the child class must have the same name, return type, and parameters as in the parent class.
- The `@Override` annotation is often used to indicate that a method is intended to override a method in the superclass.
- The access level of the overriding method cannot be more restrictive than that of the method in the superclass.
- Only instance methods can be overridden; static methods belong to the class, not instances, and hence cannot be overridden but can be hidden.

Example of Runtime Polymorphism

```
class Animal {  
    void sound() {  
        System.out.println("Animal makes a sound");  
    }  
}
```

```
class Dog extends Animal {  
    @Override  
    void sound() {  
        System.out.println("Dog barks");  
    }  
}
```

```
class Cat extends Animal {  
    @Override  
    void sound() {  
        System.out.println("Cat meows");  
    }  
}
```

```
public class TestPolymorphism {  
    public static void main(String[] args) {  
        Animal myAnimal;  
  
        myAnimal = new Dog();  
        myAnimal.sound(); // Outputs: Dog barks  
  
        myAnimal = new Cat();  
        myAnimal.sound(); // Outputs: Cat meows  
    }  
}
```

Output:

Dog barks
Cat meows

In this example, the `sound()` method is overridden in both the `Dog` and `Cat` classes. When an `Animal` reference variable points to a `Dog` object, the `sound()` method of the `Dog` class is invoked. Similarly, when the same reference points to a `Cat` object, the `sound()` method of the

Cat class is invoked. This behavior demonstrates runtime polymorphism, where the method call is resolved based on the actual object type at runtime.

- **Dynamic Method Dispatch**

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at runtime rather than compile-time. It is the backbone of runtime polymorphism in Java. This mechanism allows Java to achieve runtime polymorphism by determining the appropriate method to execute based on the actual object type that the reference variable is pointing to and complete idea is described in Figure 8.3.

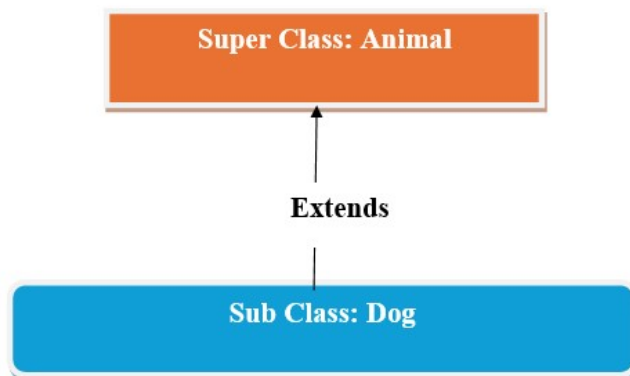


Fig 8.3 Dynamic Dispatch Method for Animal -Dog Relation

Example of Dynamic Method Dispatch:

```
class Animal {  
    void sound() {  
        System.out.println("Animal sound");  
    }  
}
```

```
class Dog extends Animal {  
    @Override  
    void sound() {  
        System.out.println("Dog barks");  
    }  
}
```

```
class TestDispatch {  
    public static void main(String[] args) {  
        Animal myAnimal = new Dog(); // Reference type is Animal, but object is Dog  
        myAnimal.sound(); // Outputs: Dog barks  
    }  
}
```

Here, although the reference variable `myAnimal` is of type `Animal`, the actual object is of type `Dog`. At runtime, the JVM determines that the `sound()` method of the `Dog` class should be called, not the `Animal` class's method. This is dynamic method dispatch in action.

Advantages of Runtime Polymorphism

1. **Flexibility and Extensibility:** Allows a program to choose the appropriate method at runtime based on the actual object, making it easier to extend and maintain.
2. **Code Reusability:** Common code can be written in the superclass, and specific behavior can be provided in subclasses, promoting reuse and reducing redundancy.
3. **Design Patterns and Frameworks:** Runtime polymorphism is fundamental to many design patterns and frameworks, enabling features like dependency injection and event handling.

Disadvantages of Runtime Polymorphism

1. **Performance Overhead:** Since method resolution occurs at runtime, there may be a slight performance overhead compared to compile-time polymorphism.
2. **Complexity in Debugging:** Debugging issues related to runtime polymorphism can be more challenging due to the dynamic nature of method invocation.

Runtime polymorphism in Java allows methods to behave differently based on the actual object at runtime. It is primarily achieved through method overriding and dynamic method dispatch, enabling flexible and extensible code. While it offers significant benefits in terms of maintainability and design, it also introduces some performance overhead and complexity. Understanding and effectively utilizing runtime polymorphism is essential for writing robust and scalable Java applications.

- **Interface Animal:** This interface declares a single method, `make Sound()`, which will be implemented by various classes.
- **Classes Dog, Cat, and Cow:** These classes implement the `Animal` interface, providing their own version of the `make Sound()` method.
- **Polymorphism in Action:** In the `Main` class, the reference `my Animal` is of type `Animal` (the interface). This reference is then pointed to different objects (`Dog`, `Cat`, and `Cow`). Even though the reference type is `Animal`, the method that gets called is determined by the actual object type that `my Animal` refers to at runtime.

Benefits of Using Interfaces for Polymorphism

1. **Flexibility:** Interfaces allow different classes to implement the same set of methods, enabling flexible and extensible designs. You can add new implementations without modifying existing code.
2. **Decoupling:** Using interfaces helps decouple code, meaning that the code using the interface doesn't need to know about the concrete classes that implement the interface.
3. **Interchangeability:** Objects of different classes can be treated as objects of a common interface type, allowing them to be used interchangeably.

4. Real-World Example

Consider a payment system where different payment methods (e.g., Credit Card, PayPal, Bank Transfer) implement a common `Payment` interface. The `Payment` interface might declare a method `process Payment()`, and each payment method class would provide its own implementation. This way, the system can process different types of payments without knowing the specifics of each payment method, achieving polymorphism.

```
interface Payment {
    void processPayment(double amount);
}

class CreditCard implements Payment {
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
    }
}

class PayPal implements Payment {
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }
}

class BankTransfer implements Payment {
    public void processPayment(double amount) {
        System.out.println("Processing bank transfer of $" + amount);
    }
}

public class PaymentProcessor {
    public void process(Payment payment, double amount) {
        payment.processPayment(amount);
    }

    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();

        Payment creditCard = new CreditCard();
        Payment payPal = new PayPal();
        Payment bankTransfer = new BankTransfer();

        processor.process(creditCard, 100.0);
        processor.process(payPal, 200.0);
        processor.process(bankTransfer, 300.0);
    }
}
```

In this example, the PaymentProcessor can process any payment method that implements the Payment interface, demonstrating the power of polymorphism through interfaces.

By leveraging interfaces, you can design systems that are modular, easy to maintain, and adaptable to change.

8.5 POLYMORPHISM WITH INTERFACES

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent class or interface. In Java, polymorphism can be achieved in several ways, one of which is through interfaces.

Understanding Polymorphism with Interfaces

❖ What is an Interface?

An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields, constructors, or method implementations (other than default methods). A class or another interface can implement an interface.

❖ Polymorphism through Interfaces

When a class implements an interface, it agrees to perform the specific behaviors defined by the interface. Polymorphism is achieved because a single action can behave differently based on the object that implements the interface.

Example: Polymorphism using Interfaces

Consider the following example where polymorphism is demonstrated using an interface.

```
// Define an interface
interface Animal {
    void makeSound();
}

// Implement the interface in different classes
class Dog implements Animal {
    public void makeSound() {
        System.out.println("Woof");
    }
}

class Cat implements Animal {
    public void makeSound() {
        System.out.println("Meow");
    }
}

class Cow implements Animal {
    public void makeSound() {
        System.out.println("Moo");
    }
}

// Demonstrate polymorphism with interfaces
public class Main {
    public static void main(String[] args) {
        // Declare an interface type reference
        Animal myAnimal;

        // Point the reference to different objects
        myAnimal = new Dog();
        myAnimal.makeSound(); // Outputs: Woof

        myAnimal = new Cat();
```

```

myAnimal.makeSound(); // Outputs: Meow

myAnimal = new Cow();
myAnimal.makeSound(); // Outputs: Moo
    }
}
    
```

8.6 POLYMORPHISM AND ABSTRACT CLASSES

Polymorphism is one of the core principles of object-oriented programming (OOP), and it allows objects to be treated as instances of their parent class or interface, rather than their actual derived class. In Java, polymorphism can also be achieved through abstract classes.

Understanding Polymorphism with Abstract Classes is shown in Figure 8.4

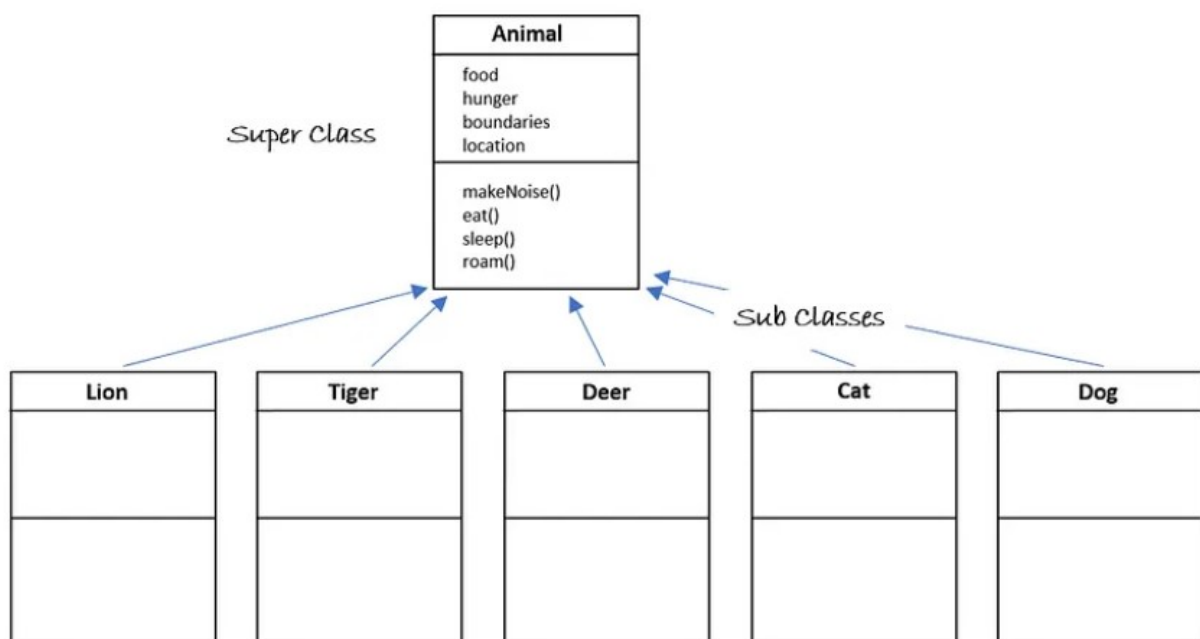


Fig 8.4. Interface and Abstract class implementation

❖ **What is an Abstract Class?**

An abstract class in Java is a class that cannot be instantiated on its own and is intended to be subclassed. It can contain abstract methods (methods without a body) as well as concrete methods (methods with a body). Abstract classes are used to define a common interface for a group of related classes while also allowing for some level of implementation reuse.

❖ **Polymorphism through Abstract Classes**

When a class inherits from an abstract class and provides implementations for the abstract methods, polymorphism is achieved because the base type (the abstract class) can be used to reference objects of the derived types.

Example: Polymorphism using Abstract Classes

Consider the following example to understand how polymorphism works with abstract classes:

```

// Define an abstract class
    
```

```
abstract class Animal {
    // Abstract method (does not have a body)
    abstract void makeNoise();

    // Concrete method
    void eat() {
        System.out.println("This animal is eating.");
    }
}

// Subclasses provide implementations for the abstract method
class Dog extends Animal {
    @Override
    void makeNoise () {
        System.out.println("Woof");
    }
}

class Cat extends Animal {
    @Override
    void makeNoise () {
        System.out.println("Meow");
    }
}

class Cow extends Animal {
    @Override
    void makeNoise () {
        System.out.println("Moo");
    }
}

// Demonstrate polymorphism with abstract classes
public class Main {
    public static void main(String[] args) {
        // Declare an abstract class type reference
        Animal myAnimal;

        // Point the reference to different objects
        myAnimal = new Dog();
        myAnimal.makeNoise (); // Outputs: Woof
        myAnimal.eat(); // Outputs: This animal is eating.

        myAnimal = new Cat();
        myAnimal.makeNoise (); // Outputs: Meow
        myAnimal.eat(); // Outputs: This animal is eating.

        myAnimal = new Cow();
        myAnimal.makeNoise (); // Outputs: Moo
        myAnimal.eat(); // Outputs: This animal is eating.
    }
}
```



```

    }
}

```

- **Abstract Class Animal:** This abstract class contains one abstract method, `makeSound()`, and one concrete method, `eat()`. The `makeSound()` method must be implemented by any subclass of `Animal`.
- **Classes Dog, Cat, and Cow:** These classes extend the `Animal` class and provide specific implementations for the `makeSound()` method.
- **Polymorphism in Action:** In the `Main` class, the reference `myAnimal` is of type `Animal` (the abstract class). This reference can be assigned to any subclass object (`Dog`, `Cat`, or `Cow`). The method `makeSound()` called on `myAnimal` is determined by the actual object type that `myAnimal` refers to at runtime.

Benefits of Using Abstract Classes for Polymorphism

1. **Shared Code and Reusability:** Abstract classes allow for code reuse by providing a base implementation for some methods while forcing subclasses to implement the abstract methods.
2. **Flexibility with Common Behavior:** Abstract classes can define common behavior (in concrete methods) that all subclasses should share, while allowing subclasses to override or provide their unique implementation of other behaviors.
3. **Ease of Extension:** New classes can be easily added to the system by extending the abstract class and providing specific implementations for abstract methods.

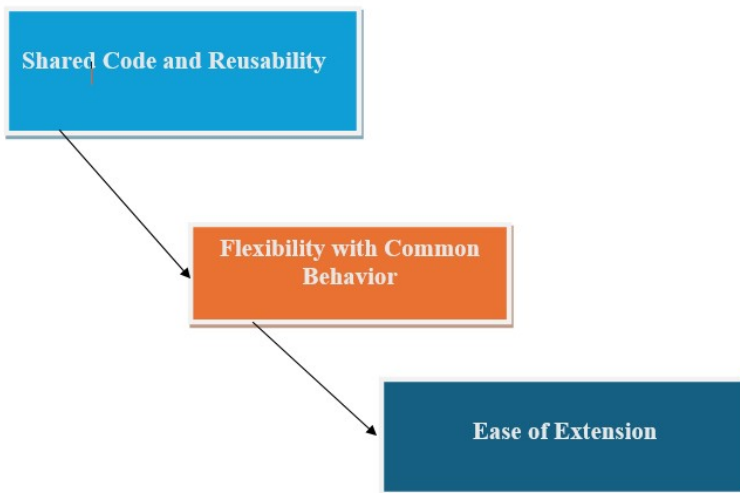


Fig 8.5 Benefits of Using Abstract Classes for Polymorphism

Real-World Example

Consider a shape system where different shapes (e.g., `Circle`, `Rectangle`, `Triangle`) inherit from an abstract class `Shape`. The `Shape` class might declare an abstract method `draw()` that each subclass must implement.

```

abstract class Shape {
    abstract void draw();

    void moveTo(int x, int y) {
        System.out.println("Moving to x: " + x + ", y: " + y);
    }
}

```

```
class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a Circle");
    }
}

class Rectangle extends Shape {
    void draw() {
        System.out.println("Drawing a Rectangle");
    }
}

class Triangle extends Shape {
    void draw() {
        System.out.println("Drawing a Triangle");
    }
}

public class ShapeDemo {
    public static void main(String[] args) {
        Shape myShape;

        myShape = new Circle();
        myShape.draw(); // Outputs: Drawing a Circle
        myShape.moveTo(5, 10); // Outputs: Moving to x: 5, y: 10

        myShape = new Rectangle();
        myShape.draw(); // Outputs: Drawing a Rectangle
        myShape.moveTo(15, 20); // Outputs: Moving to x: 15, y: 20

        myShape = new Triangle();
        myShape.draw(); // Outputs: Drawing a Triangle
        myShape.moveTo(25, 30); // Outputs: Moving to x: 25, y: 30
    }
}
```

In this example, the Shape abstract class provides a common interface and some shared functionality (moveTo method), while specific shapes like Circle, Rectangle, and Triangle provide their own implementation of the draw() method. The abstract class enables polymorphism, allowing the Shape reference to be used interchangeably for any shape subclass.

Key Points

- Abstract classes are useful when you have a base class that should not be instantiated and should define a common interface for its subclasses.
- Polymorphism allows for dynamic method dispatch, where the method that gets executed is determined at runtime based on the actual object's type, not the reference's type.
- Abstract classes can have both abstract methods (which must be implemented by subclasses) and concrete methods (which can be shared across all subclasses).

Using abstract classes for polymorphism is a powerful tool in Java, especially when you need to define a common behavior across multiple classes while still allowing each class to provide its unique implementation.

8.7 ADVANTAGES AND DISADVANTAGES OF POLYMORPHISM

8.7.1 Advantages

- **Flexibility:** Easier to introduce new implementations.
- **Code Reusability:** Reduces code duplication.
- **Ease of Maintenance:** Changes in code are localized.

8.7.2 Disadvantages

- **Complexity:** Can introduce complexity and make debugging harder.
- **Performance:** Dynamic method dispatch can have a slight performance overhead.

8.8 COMMON MISTAKES AND BEST PRACTICES

8.8.1 Common Mistakes

- Confusing method overloading with overriding.
- Misusing polymorphism, leading to overly complex hierarchies.

8.8.2 Best Practices

- Favor composition over inheritance where possible.
- Keep class hierarchies shallow.
- Use `@Override` annotations to avoid accidental overloading.

8.9 SUMMARY

Polymorphism is a fundamental concept in Java's object-oriented programming that enables objects to be treated as instances of their parent class or interface, allowing for more flexible and scalable code. It allows a single interface or abstract class to be used for a general class of actions, while specific behavior is determined by the actual subclass or implementation at runtime. This is achieved through method overriding in subclasses or through the implementation of interfaces. Polymorphism promotes code reusability, modularity, and makes it easier to manage and extend applications. By utilizing polymorphism, developers can design systems that are more adaptable to change, maintainable, and robust, as it decouples the code that uses the polymorphic objects from the specific implementation details of those objects.

8.10 TECHNICAL TERMS

- Polymorphism
- Abstract Class
- Interface
- Compile Time
- Run Time
- Overriding

8.11 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Discuss how compile-time polymorphism (method overloading) and runtime polymorphism (method overriding) are implemented in Java.
2. Compare and contrast the use of interfaces and abstract classes in achieving polymorphism in Java.
3. How does method overriding affect exception handling, particularly with checked and unchecked exceptions?

Short questions:

1. What is Polymorphism in Java?
2. How is Polymorphism Achieved in Java?
3. What is the Difference Between Overloading and Overriding in the Context of Polymorphism?
4. How Does Polymorphism Work with Interfaces and Abstract Classes?
5. What are the Advantages and Disadvantages of Polymorphism?

8.12 SUGGESTED READINGS

1. "Java: The Complete Reference" by Herbert Schildt, 12th Edition (2021), McGraw-Hill Education
2. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition (2005), O'Reilly Media
3. "Effective Java" by Joshua Bloch, 3rd Edition (2018), Addison-Wesley Professional

Dr. KAMPA LAVANYA

LESSON- 9

PACKAGES

OBJECTIVES:

After going through this lesson, you will be able to

- Describe the role of packages in organizing and managing classes and interfaces
- Distinguish between built-in packages and user-defined packages.
- Write the syntax for declaring a package and explain the naming conventions for packages.
- Describe how to run Java programs that include classes from user-defined packages.
- Explain how packages and sub-packages help in avoiding naming conflicts

STRUCTURE:

- 9.1 Java package
 - 9.1.1 Key Benefits of Using Packages
- 9.2 Types of packages
 - 9.2.1 Built-in Packages
 - 9.2.2 User-defined Packages
- 9.3 Creating and running a package
 - 9.3.1 Creating a Package
 - 9.3.2 Steps to Create a Package:
- 9.4 Compiling and running a packages
- 9.5 Accessing a package
 - 9.5.1 Using packagename
 - 9.5.2 Using packagename.classname
 - 9.5.3 Using fully qualified name
- 9.6 Sub package
 - 9.6.1 Creating Sub-packages
 - 9.6.2 Compiling and Running Classes in Sub-packages
 - 9.6.3 Accessing Sub-packages
- 9.7 Summary
- 9.8 Technical Terms
- 9.9 Self-Assessment Questions
- 9.10 Further Readings

9.1 JAVA PACKAGE

Packages in Java are a mechanism to group related classes, interfaces, and sub-packages together. This helps in organizing files within a project, avoiding naming conflicts, and controlling access to classes and interfaces. A package acts like a folder in a file system and can contain classes, interfaces, sub-packages, and other packages.

9.1.1 Key Benefits of Using Packages

1. **Namespace Management:** Packages prevent naming conflicts by differentiating classes and interfaces with the same name but in different packages.

2. **Access Protection:** Packages allow control over the accessibility of classes and interfaces. Members with default (package-private) access are accessible only within their own package.
3. **Code Organization and Modularity:** Packages help in organizing classes logically, making code easier to manage, maintain, and understand.
4. **Reusability:** By grouping related classes and interfaces, packages promote reusability of code across different projects.

9.2 TYPES OF PACKAGES

Java provides two main types of packages:

1. **Built-in Packages:** These are pre-defined packages that come with the Java Standard Library. Examples include 'java.lang', 'java.util', 'java.io', and 'java.awt'.
2. **User-defined Packages:** These are custom packages created by the programmer to group related classes and interfaces based on the functionality of the application.

9.2.1 Built-in Packages

Built-in packages provide a large set of reusable classes for various functionalities:

- 'java.lang': Contains fundamental classes such as 'String', 'System', and 'Math'. This package is automatically imported by the compiler for every Java program.
- 'java.util': Provides utility classes like 'ArrayList', 'HashMap', 'Date', and many others for data structure management, date manipulation, and more.
- 'java.io': Contains classes for input and output operations, such as 'File', 'FileReader', 'BufferedReader', and 'PrintWriter'.
- 'java.awt': Includes classes for building graphical user interface (GUI) components, like 'Button', 'Frame', and 'Canvas'.

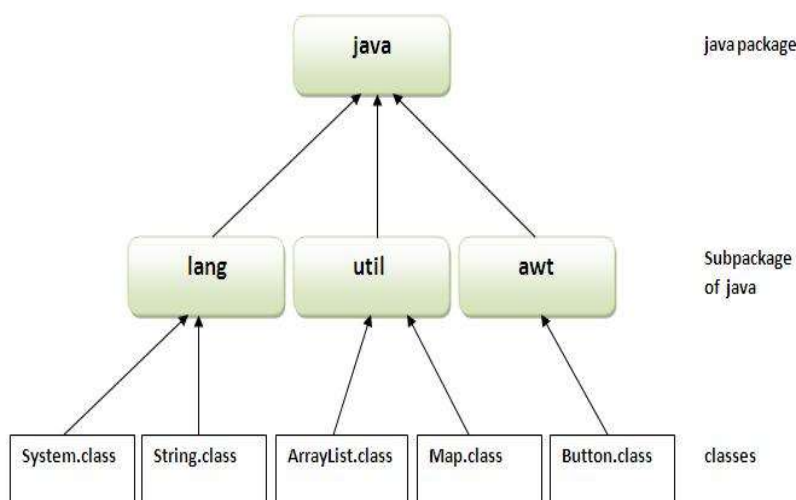


Figure 9.1 built in packages

9.2.2 User-defined Packages

User-defined packages are created by developers to encapsulate their classes and interfaces. This is especially useful for larger projects where different modules may need to be developed and maintained separately.

9.3 CREATING AND ACCESSING A PACKAGE

9.3.1 Creating a Package

To create a package in Java, you need to declare the package name at the very top of your Java source file, before any 'import' statements or class definitions.

Syntax:

```
package packageName;
```

Example:

```
// File: MyPack/Car.java
package MyPack;

public class Car {
    public void display() {
        System.out.println("This is a car.");
    }
}
```

In this example, we create a package named "MyPack" and define a "Car" class inside it.

9.3.2 Steps to Create a Package:

1. **Choose a Package Name:** The package name should be unique to avoid conflicts and should follow the naming conventions (usually lowercase and reflective of the functionality or company domain).
2. **Declare the Package:** At the top of your Java file, use the 'package' keyword followed by the package name.
3. **Save the File:** Save the Java file in a directory structure that matches the package name. For example, "MyPack.Car" should be saved in a directory named "MyPack".

9.4 COMPILING AND RUNNING THE PACKAGE

To compile the class inside the package, navigate to the source directory and use the 'javac' command with the full path to the file.

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For example

```
javac -d . package/javafile.java
```

The -d switch specifies the destination where to put the generated class file.

We can use any directory If you want to keep the package within the same directory, you can use . (dot).

```
javac -d . Mypack/Car.java
```

To run a class from a package, use the 'java' command with the fully qualified class name (including the package name).

Example:

```
java MyPack.Car
```

We need to use fully qualified name e.g. *mypack.Simple* etc to run the class.

To Compile: javac -d . Car.java

To Run: java mypack.Car

Output: This is a Car

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The • (dot) represents the current folder.

9.5 ACCESSING A PACKAGE

There are three ways to access the package from outside the package.

- Import package.*;
- import package.classname;
- fully qualified name.

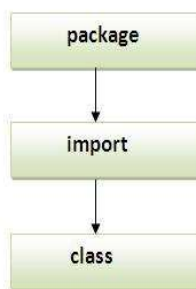


Figure 9.2 package accessing hierarchy

9.5.1 Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages. The import keyword is used to make the classes and interface of another package accessible to the current package.

Program 1:

```
//save by A.java Javac -d . A.java
package pack;
public class A
{
    public void msg()
    {
        System.out.println("Hello");
    }
}
```

Program 2:

```
//save by B.java
package mypack;
import pack.*;
```



```
class B
{
    public static void main(String args[])
    {
        A obj = new A();
        obj.msg();
    }
}
```

```
Javac -d B.java
Java mypack.B
```

9.5.2 Using packagename.classname

If we import *package.classname* then only declared class of this package will be accessible.

```
//save by A.java
```

```
package pack;
public class A
{
    public void msg()
    {
        System.out.println("Hello");
    }
}
```

```
//save by B.java
```

```
package mypack;
import pack.A;

class B
{
    public static void main(String args[])
    {
        A obj = new A();
        obj.msg();
    }
}
```

9.5.3 Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class

//save by A.java

```
package pack;
public class A
{
    public void msg()
    {
        System.out.println("Hello");
    }
}
```

```
}

//save by B.java

package mypack;
import pack.*;

class B
{
    public static void main(String args[])
    {
        pack.A obj = new pack.A();
        obj.msg();
    }
}
```

Note: If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

9.6 SUB PACKAGE

A sub-package in Java is a package that is nested within another package. It is essentially a package inside another package and helps in further organizing classes and interfaces in a hierarchical manner.

Sub-packages allow developers to create a more structured organization of related classes and interfaces by grouping them into a hierarchy of packages. This is useful in large projects where grouping related functionalities together in a clear structure is important. Sub-packages are named by adding another level to the existing package name, separated by a dot ('.'). For example, if you have a main package named 'com.example', a sub-package might be 'com.example.utils'.

In Java, sub-packages do not inherit access privileges from their parent packages. Each sub-package is treated as an independent package, even though they are hierarchically related. This means that the classes and interfaces in a sub-package are not automatically accessible to the parent package, unless explicitly imported.

9.6.1 Creating Sub-packages

To create a sub-package, you simply declare it by specifying the full package name at the beginning of your Java source file. This includes the parent package and the sub-package.

Example:

Let's create a package 'com.example' with a sub-package 'com.example.utils'.

1. Main Package: 'com.example'

```
// File: com/example/Car.java
package com.example;

public class Car {
    public void display() {
        System.out.println("This is a car from com.example package.");
    }
}
```

2. Sub-package: 'com.example.utils'

```
// File: com/example/utils/Helper.java
package com.example.utils;

public class Helper {
    public void help() {
        System.out.println("Helper class in com.example.utils
package.");
    }
}
```

9.6.2 Compiling and Running Classes in Sub-packages

To compile classes that belong to a sub-package, you should maintain the directory structure that reflects the package name.

Compilation:

```
javac com/example/Car.java
javac com/example/utils/Helper.java
```

Running:

To run a class from the sub-package, you need to provide the fully qualified class name:

```
java com.example.Car
java com.example.utils.Helper
```

9.6.3 Accessing Sub-packages

To access a class or interface from a sub-package in another Java file, you need to import it using the 'import' statement with the full package name.

Example:

```
import com.example.utils.Helper;
public class Main {
    public static void main(String[] args) {
        Helper helper = new Helper();
        helper.help();    // Output: Helper class in com.example.utils
package.
    }
}
```

Alternatively, you can use a wildcard to import all classes from the sub-package:

```
import com.example.utils.*;

public class Main {
    public static void main(String[] args) {
        Helper helper = new Helper();
        helper.help();    // Output: Helper class in com.example.utils
package.
    }
}
```

Access Modifiers ->	Most Restrictive ← → Least Restrictive			
	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y

Figure 9.3 package accessing level

9.7 SUMMARY

The chapter on Java packages introduces the concept of packages, which are used to group related classes and interfaces to manage namespaces, promote modularity, and enhance code organization. It covers the two main types of packages: built-in packages provided by the Java Standard Library and user-defined packages created by developers for custom use.

The chapter explains how to create a package, compile and run classes within it, and access classes from a package using the `import` statement. It also discusses sub-packages, which are packages nested within other packages, allowing for a hierarchical structure that further organizes code into logical groups. Through this, the chapter emphasizes the importance of packages in maintaining clean and manageable Java projects.

9.8 TECHNICAL TERMS

Package, modularity, hierarchy, sub package.

9.9 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Describe the purpose of packages in Java . Illustrate the types of packages with examples.
2. Explain the steps involved in creating, compiling, and running a package in Java.
3. Discuss the concept of sub-packages in Java.
4. What are the different ways to access a class from a package in Java?

Short Answer Questions:

1. What is a package in Java, and why is it important?
2. List two types of packages in Java and provide examples for each.
3. How do you create a package in Java? Explain with syntax.
4. Explain how to access a class from a package in another Java file.

9.10 SUGGESTED READINGS

- 1) Herbert Schildt and Dale Skrien “Java Fundamentals –A comprehensive Introduction”, McGraw Hill, 1st Edition, 2013.
- 2) Herbert Schildt, “Java the complete reference”, McGraw Hill, Osborne, 11th Edition, 2018.
- 3) T. Budd “Understanding Object-Oriented Programming with Java”, Pearson Education, Updated Edition (New Java 2 Coverage), 1999 REFERENCE BOOKS:
- 4) P.J. Dietel and H.M. Dietel “Java How to program”, Prentice Hall, 6th Edition, 2005.
- 5) P. Radha Krishna “Object Oriented programming through Java”, CRC Press, 1st Edition, 2007.
- 6) Malhotra and S. Choudhary “Programming in Java”, Oxford University Press, 2nd Edition, 2014

Dr. U. SURYA KAMESWARI

LESSON- 10

FILES

OBJECTIVES:

After going through this lesson, you will be able to

- Learn about the concept of streams in Java
- Understand the different constructors of `FileOutputStream` and `FileInputStream` and their usage.
- Understand the use of different constructors of `FileWriter` for creating or appending to files.
- Understand how to create and use `FileReader` to read character data from files.
- Apply the knowledge of stream classes to solve real-world file input and output problems.

STRUCTURE:

- 10.1 Stream classes
 - 10.1.1 Byte Streams
 - 10.1.2 Character Streams
- 10.2 Creating a File using File Output Stream
 - 10.2.1 Steps to Create a File Using `FileOutputStream`
 - 10.2.2 Example: Creating a File Using `FileOutputStream`
- 10.3 Reading Data from a File using File Input Stream
 - 10.3.1 Steps to Read Data from a File Using `FileInputStream`
 - 10.3.2 Example: Reading Data from a File Using `FileInputStream`
- 10.4 Creating a File using File Writer
 - 10.4.1 Steps to Create a File Using `FileWriter`
 - 10.4.2 Example: Creating a File Using `FileWriter`
- 10.5 Reading a File using File Reader
 - 10.5.1 Steps to Read a File Using `FileReader`
 - 10.5.2 Example: Reading a File Using `FileReader`
- 10.6 Summary
- 10.7 Technical Terms
- 10.8 Self-Assessment Questions
- 10.9 Further Readings

10.1 STREAM CLASSES

Java streams provide a way to handle input and output operations (I/O) in Java. They enable reading data from a source or writing data to a destination in a sequential manner.

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices `STDIN`, `STDOUT` and `STDERR`. Similar way Java provides following three standard streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as System.out.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as System.err.

No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are basically the same

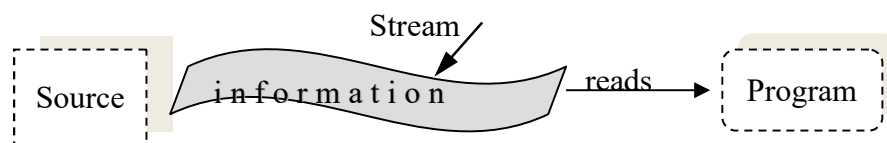


Figure 10.1 read stream

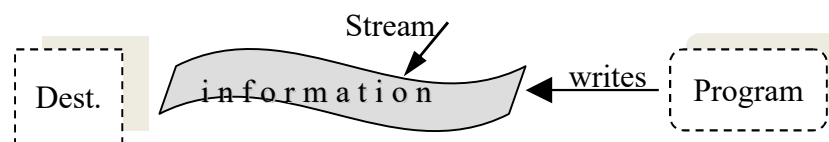


Figure 10.2 Write stream

Java provides two primary types of streams:

1. Byte Streams: Used for reading and writing binary data (8-bit bytes).
2. Character Streams: Used for reading and writing text data (16-bit Unicode characters).

10.1.1 Byte Streams

A **byte stream** in Java is a type of stream used to perform input and output operations of raw binary data, which is represented in 8-bit bytes. Byte streams are primarily used for reading and writing binary data, such as images, audio files, and other non-text data types. They are part of Java's I/O (Input/Output) system, which is used to handle data streams for reading and writing data to and from files, network connections, or other input/output sources.

Features of Byte Streams

- **Handles Raw Binary Data:** Byte streams are designed to handle raw binary data, which makes them suitable for files and data sources that are not in a human-readable text format.
- **8-Bit Bytes:** Byte streams operate on 8-bit bytes, which means they process data one byte at a time. This is ideal for data that is already in byte format or needs to be processed at the byte level.
- **Unbuffered and Buffered Streams:** Byte streams come in both unbuffered and buffered variants. Unbuffered streams handle each byte individually, while buffered streams use an internal buffer to optimize read and write operations by reducing the number of native I/O calls.

- **Used for Binary Files:** Byte streams are commonly used to read and write binary files like images, audio files, and serialized objects, where precise control over the binary format is required.

Java provides several classes for handling byte streams, but the most commonly used are:

InputStream: The base class for all byte input streams in Java. It defines methods for reading bytes from a source.

OutputStream: The base class for all byte output streams in Java. It defines methods for writing bytes to a destination.

Some of the specific subclasses of `InputStream` and `OutputStream` include:

FileInputStream: A subclass of `InputStream` used for reading bytes from a file.

FileOutputStream: A subclass of `OutputStream` used for writing bytes to a file.

BufferedInputStream: A subclass of `InputStream` that adds buffering to improve reading performance by reducing the number of native I/O operations.

BufferedOutputStream: A subclass of `OutputStream` that adds buffering to improve writing performance by reducing the number of native I/O operations.

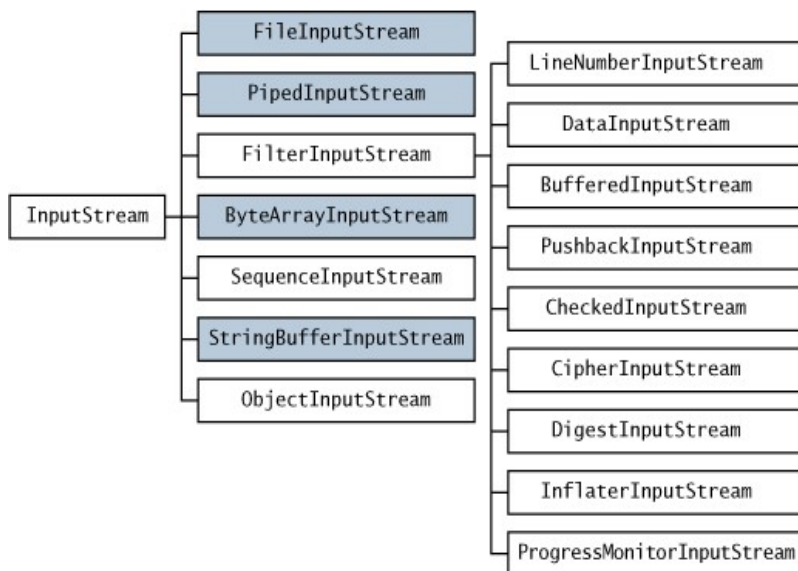
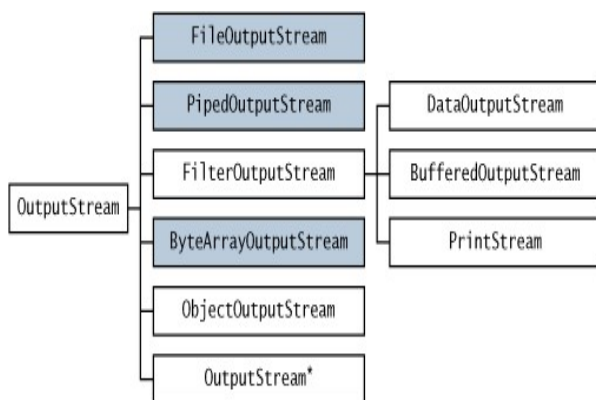


Figure 10.3 input stream class hierarchy



* In a different package

Figure 10.4 outputstream class hierarchy

10.1.2 Character Streams

A **character stream** in Java is a type of stream used to handle input and output operations of character data. Character streams are designed to work with data in a human-readable text format, such as Unicode characters. These streams are ideal for processing text files or any data source where the input and output are in character form rather than raw binary data.

Features of Character Streams

- **Handles Character Data:** Character streams work with 16-bit Unicode characters, making them suitable for reading and writing text data. This includes letters, digits, and other textual symbols.
- **Automatic Character Encoding and Decoding:** Character streams automatically handle character encoding and decoding, making it easier to work with text files in different languages and character sets.
- **Buffered and Unbuffered Streams:** Similar to byte streams, character streams also come in buffered and unbuffered variants. Buffered streams provide higher performance by minimizing the number of I/O operations through the use of an internal buffer.
- **Suitable for Text Files:** Character streams are commonly used for reading from and writing to text files, where the data is encoded in character format rather than binary.

Character Stream Classes in Java

Java provides several classes for handling character streams. The two main abstract base classes for character streams are:

Reader: The base class for all character input streams in Java. It defines methods for reading character data.

Writer: The base class for all character output streams in Java. It defines methods for writing character data.

Some of the specific subclasses of `Reader` and `Writer` include:

FileReader: A subclass of `Reader` used for reading characters from a file.

FileWriter: A subclass of `Writer` used for writing characters to a file.

BufferedReader: A subclass of `Reader` that adds buffering to improve reading performance by reducing the number of native I/O operations.

BufferedWriter: A subclass of `Writer` that adds buffering to improve writing performance by reducing the number of native I/O operations.

InputStreamReader: A bridge from byte streams to character streams; reads bytes and decodes them into characters using a specified charset.

OutputStreamWriter: A bridge from character streams to byte streams; encodes characters into bytes using a specified charset.

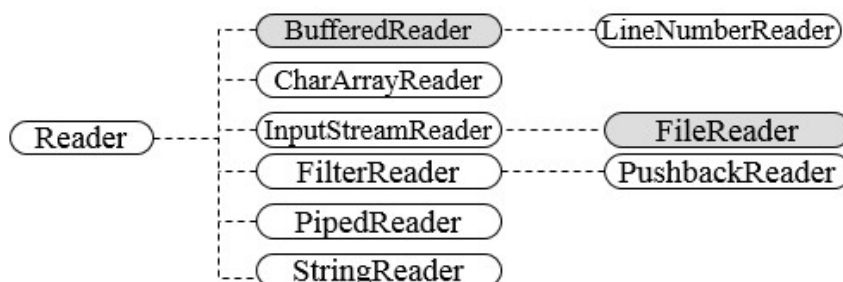


Figure 10.6 Reader class stream hierarchy

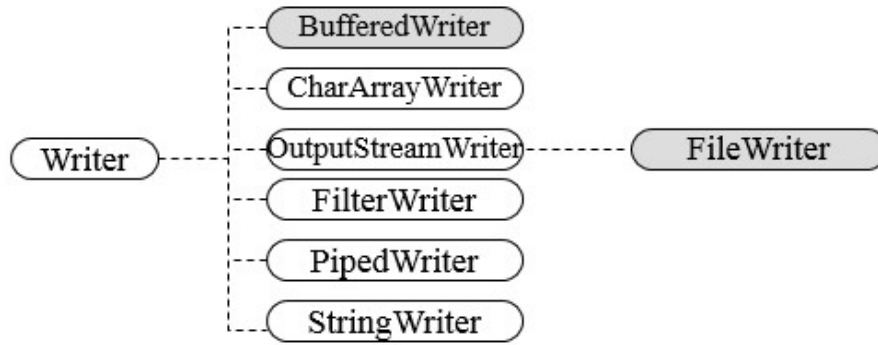


Figure 10.6 Writer class stream hierarchy

10.2 CREATING A FILE USING FILE OUTPUT STREAM

Creating a file using `FileOutputStream` in Java involves writing raw byte data to a file. The `FileOutputStream` class is part of the byte stream family, which is designed for handling raw binary data. It can be used to create a file and write data to it byte by byte.

10.2.1 Steps to Create a File Using `FileOutputStream`

- **Import the Necessary Package:** `FileOutputStream` is part of the `java.io` package, so you need to import it.
- **Create an Instance of `FileOutputStream`:** You need to create a `FileOutputStream` object, specifying the file you want to create or write to. If the file doesn't exist, it will be created. If it exists, it can either be overwritten or appended to, depending on the constructor used.
- **Write Data to the File:** Use the `write()` method to write data to the file. The data should be in the form of bytes, so if you're writing text, you'll need to convert it into bytes using `String.getBytes()`.
- **Close the Stream:** Always close the `FileOutputStream` using the `close()` method to release system resources and ensure all data is properly written to the file.

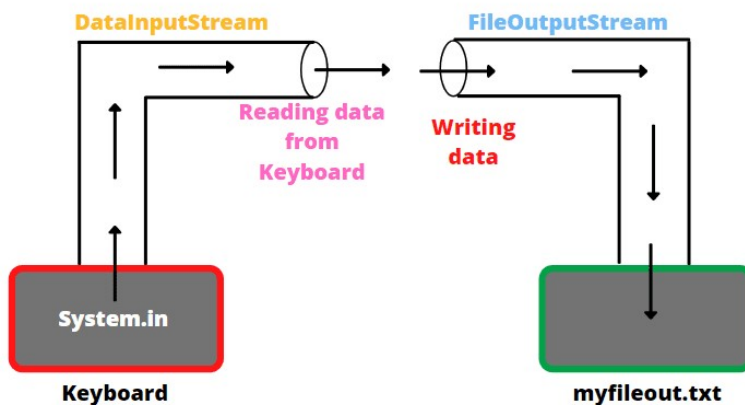


Figure 10.7: Creating a text file

10.2.2 Example: Creating a File Using `FileOutputStream`

```
import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutputStreamExample {
    public static void main(String[] args) {
        String data = "This is an example of writing data to a file using
FileOutputStream.";

        // Try-with-resources statement ensures that each resource is
        closed at the end of the statement
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            // Convert string data to byte array
            byte[] byteData = data.getBytes();

            // Write byte array to the file
            fos.write(byteData);

            // Output a message indicating success
            System.out.println("Data successfully written to the file.");
        } catch (IOException e) {
            // Handle any IOExceptions that may occur
            e.printStackTrace();
        }
    }
}
```

The following steps gives explanation for the above program..

- **String data:** A string containing the data to be written to the file.
- **FileOutputStream fos = new FileOutputStream("output.txt"):** This line creates a new `FileOutputStream` object to write to a file named "output.txt". If the file does not exist, it will be created. If it exists, it will be overwritten (if you want to append to the file instead, use `new FileOutputStream("output.txt", true)`).
- **data.getBytes():** Converts the string `data` to a byte array. The `write()` method of `FileOutputStream` requires data in byte format.
- **fos.write(byteData):** Writes the byte array to the file.
- **fos.close():** Closes the `FileOutputStream`. The `try-with-resources` statement ensures that the stream is automatically closed, even if an exception occurs.
- **Using try-with-resources :** In the example above, we use the `try-with-resources` statement, which is a good practice when dealing with I/O streams. This statement automatically closes the stream after the try block has finished executing, which helps to avoid resource leaks and ensures that the file is properly closed.

10.3 READING DATA FROM A FILE USING FILE INPUT STREAM

Reading data from a file using `FileInputStream` in Java involves opening a file and reading its raw byte data. The `FileInputStream` class is part of Java's I/O (Input/Output) system and is used for reading streams of raw bytes, such as image or audio files. It's especially useful when dealing with binary files where the data is not in a human-readable format.

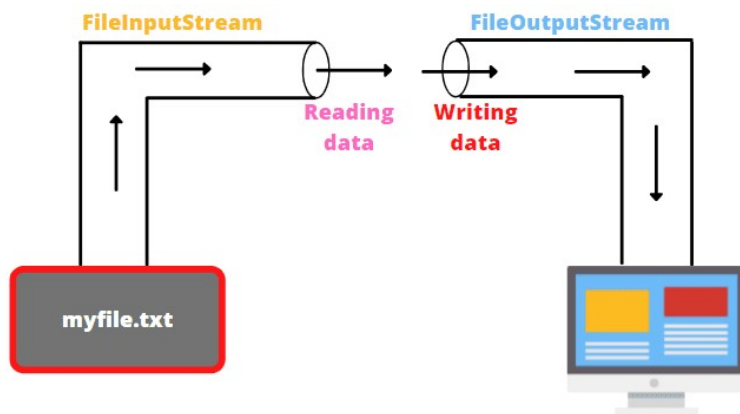


Figure 10.8 : Reading data from a text file

10.3.1 Steps to Read Data from a File Using `FileInputStream`

- **Import the Necessary Package:** `FileInputStream` is part of the `java.io` package, so you need to import it.
- **Create an Instance of `FileInputStream`:** Create a `FileInputStream` object by passing the path of the file you want to read to its constructor. This opens the file for reading.
- **Read Data from the File:** Use the `read()` method to read data from the file. This method reads the next byte of data and returns it as an `int`. If the end of the file is reached, it returns `-1`.
- **Close the Stream:** Always close the `FileInputStream` using the `close()` method to free up system resources.

10.3.2 Example: Reading Data from a File Using `FileInputStream`

```
import java.io.FileInputStream;
import java.io.IOException;

public class FileInputStreamExample {
    public static void main(String[] args) {
        // Specify the file path
        String filePath = "example.txt";

        // Try-with-resources statement to ensure the FileInputStream is
        // closed
        try (FileInputStream fis = new FileInputStream(filePath)) {
            // Variable to hold the byte being read
            int byteData;

            // Read until the end of the file
            while ((byteData = fis.read()) != -1) {
                // Convert byte data to character and print to console
                System.out.print((char) byteData);
            }
        } catch (IOException e) {
            // Handle any IOExceptions
            e.printStackTrace();
        }
    }
}
```

The following steps gives explanation for the above program..

- **String filePath = "example.txt":** Defines the path to the file that will be read. In this example, it assumes the file is in the current working directory.
- **FileInputStream fis = new FileInputStream(filePath):** Creates a new `FileInputStream` object for the specified file. If the file does not exist, this will throw a `FileNotFoundException`.
- **int byteData:** A variable to store the data read from the file. The `read()` method returns the next byte of data as an `int`. If the end of the file is reached, `read()` returns `-1`.
- **while ((byteData = fis.read()) != -1):** This loop reads the file byte by byte until the end of the file is reached. Each byte read is cast to a `char` and printed to the console.
- **fis.close():** Although we do not explicitly call `close()` here, the `try-with-resources` statement ensures that the `FileInputStream` is closed automatically when the `try` block is exited, either normally or because of an exception.
- **Using try-with-resources:** The example uses the `try-with-resources` statement, which is a recommended practice when working with I/O streams in Java. This statement automatically closes the stream when it is no longer needed, helping to prevent resource leaks.

10.4 CREATING A FILE USING FILE WRITER

Creating a file using `FileWriter` in Java involves writing character data to a file. `FileWriter` is part of Java's character stream classes, which are used for handling text data. It writes characters to a file in a platform-independent manner, making it suitable for working with text files.

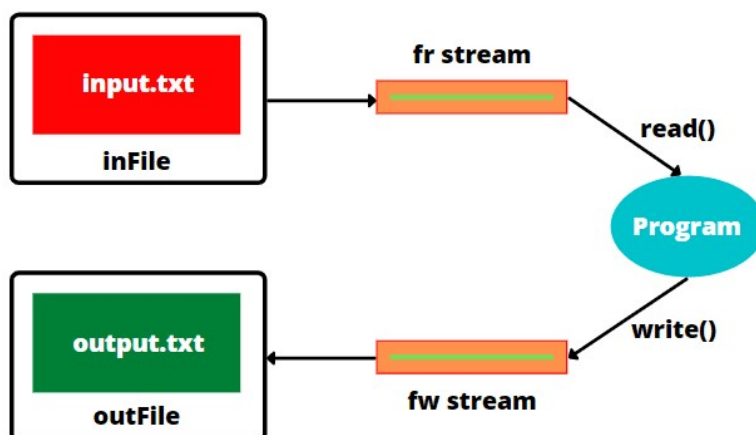


Figure 10.9 Reading from and writing to files

10.4.1 Steps to Create a File Using `FileWriter`:

- **Import the Necessary Package:** `FileWriter` is part of the `java.io` package, so you need to import it.
- **Create an Instance of `FileWriter`:** Create a `FileWriter` object by specifying the file you want to create or write to. If the file doesn't exist, `FileWriter` will create it. If it exists, it will be overwritten by default (you can append to the file by passing a second argument as `true`).

- **Write Data to the File:** Use the `write()` method to write data to the file. The data should be in the form of a string or an array of characters.
- **Close the Stream:** Always close the `FileWriter` using the `close()` method to ensure that all data is properly written and resources are released.

10.4.2 Example: Creating a File Using `FileWriter`

```
import java.io.FileWriter;
import java.io.IOException;

public class FileWriterExample {
    public static void main(String[] args) {
        String data = "This is an example of writing data to a file using
FileWriter.";

        // Using try-with-resources to ensure FileWriter is closed
        try (FileWriter fw = new FileWriter("output.txt")) {
            // Write data to the file
            fw.write(data);

            // Print confirmation message
            System.out.println("Data successfully written to the file.");
        } catch (IOException e) {
            // Handle any IOExceptions
            e.printStackTrace();
        }
    }
}
```

The following steps gives explanation for the above program.

- **`FileWriter fw = new FileWriter("output.txt", true)`:** The second argument (`true`) tells the `FileWriter` to append to the file instead of overwriting it. If the file doesn't exist, it will be created.
- **`fw.write(data)`:** This writes the string `data` to the file. Since the `FileWriter` is in append mode, the data is added to the end of the file
- **Using `try-with-resources`:** In both examples, we use the `try-with-resources` statement, which is a best practice when dealing with I/O streams in Java. This statement ensures that the `FileWriter` is automatically closed, even if an exception occurs, which helps to prevent resource leaks.

10.5 READING A FILE USING FILE READER

Reading a file using `FileReader` in Java involves reading character data from a file. `FileReader` is part of Java's character stream classes, designed for reading streams of characters from a file. It's particularly useful for handling text files where the data is in human-readable format.

10.5.1 Steps to Read a File Using `FileReader`

- **Import the Necessary Package:** `FileReader` is part of the `java.io` package, so you need to import it.
- **Create an Instance of `FileReader`:** Create a `FileReader` object by passing the file name or `File` object that you want to read from.

- **Read Data from the File:** Use the `read()` method to read characters from the file. This method reads a single character at a time and returns it as an `int`. If the end of the file is reached, it returns `-1`.
- **Close the Stream:** Always close the `FileReader` using the `close()` method to release system resources.

10.5.2 Example: Reading a File Using `FileReader`

```
import java.io.FileReader;
import java.io.IOException;

public class FileReaderExample {
    public static void main(String[] args) {
        // Specify the file to be read
        String filePath = "example.txt";

        // Using try-with-resources to ensure FileReader is closed
        try (FileReader fr = new FileReader(filePath)) {
            int character;

            // Read characters one by one from the file
            while ((character = fr.read()) != -1) {
                System.out.print((char) character);    // Print each
                character to the console
            }
            } catch (IOException e) {
                // Handle any IOExceptions
                e.printStackTrace();
            }
        }
    }
}
```

The following steps gives explanation for the above program..

- **`String filePath = "example.txt"`:** Specifies the path to the file that will be read. In this example, it assumes the file is in the current working directory.
- **`FileReader fr = new FileReader(filePath)`:** Creates a `FileReader` object for the specified file. If the file does not exist, this will throw a `FileNotFoundException`.
- **`int character`:** A variable to store each character read from the file. The `read()` method returns the next character as an `int`. If the end of the file is reached, `read()` returns `-1`.
- **`while ((character = fr.read()) != -1)`:** This loop reads the file character by character until the end of the file is reached. Each character read is cast to a `char` and printed to the console.
- **`fr.close()`:** Although not explicitly called in this example, the `try-with-resources` statement automatically closes the `FileReader` when the `try` block is exited, either normally or due to an exception.

10.6 SUMMARY

The chapter on Java Streams provides an in-depth exploration of Java's input and output (I/O) capabilities using streams. It begins by introducing the concept of stream classes, explaining the distinction between byte streams and character streams, and their respective uses for handling raw binary and text data. The chapter then delves into practical file operations, demonstrating how to create files using `FileOutputStream` and `FileWriter` for

writing byte and character data, respectively. It also covers reading files using `FileInputStream` and `FileReader`, showing how to efficiently read data from files, manage resources, and handle exceptions properly. By the end of the chapter, learners will have a comprehensive understanding of how to perform basic file I/O operations in Java, leveraging the appropriate stream classes based on the type of data being processed.

10.7 TECHNICAL TERMS

Stream, InputOutput, Byte stream, Character stream, File

10.8 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain the concept of Java streams and their hierarchy. Discuss the differences between `InputStream`, `OutputStream`, `Reader`, and `Writer` classes,
2. Discuss the process of creating and writing to a file using `FileOutputStream` and `FileWriter` in Java.
3. Describe the steps and methods involved in reading data from a file using `FileInputStream` and `FileReader`.
4. Compare these two classes in terms of their functionality, data handling, and typical use cases. Include examples to illustrate your points.
5. How do `FileWriter` and `FileReader` handle text data differently from `FileOutputStream` and `FileInputStream`? Include examples of writing and reading text files.

Short Answer Questions:

1. What is a stream in Java, and why is it important for input and output operations?
2. Differentiate between byte streams and character streams in Java. Give examples of classes used for each type of stream.
3. What is the primary purpose of the `FileOutputStream` class in Java? How do you use it to create a new file?
4. Explain how to read data from a file using `FileInputStream` in Java. What method is commonly used for this purpose?
5. Describe the purpose of the `FileWriter` class in Java. How does it differ from `FileOutputStream`?
6. How does the `FileReader` class work in Java? What kind of data is it best suited for?

10.9 SUGGESTED READINGS

- 1) Herbert Schildt and Dale Skrien “Java Fundamentals –A comprehensive Introduction”, McGraw Hill, 1st Edition, 2013.
- 2) Herbert Schildt, “Java the complete reference”, McGraw Hill, Osborne, 11th Edition, 2018.
- 3) T. Budd “Understanding Object-Oriented Programming with Java”, Pearson Education, Updated Edition (New Java 2 Coverage), 1999 REFERENCE BOOKS:
- 4) P.J. Dietel and H.M. Dietel “Java How to program”, Prentice Hall, 6th Edition, 2005.
- 5) P. Radha Krishna “Object Oriented programming through Java”, CRC Press, 1st Edition, 2007.
- 6) Malhotra and S. Choudhary “Programming in Java”, Oxford University Press, 2nd Edition, 2014

LESSON- 11

EXCEPTIONAL HANDLING

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the different types of errors
- Learn the difference between checked and unchecked exceptions.
- Learn how to use try, catch, finally, and try-with-resources statements to handle exceptions effectively.
- Understand the use of the throws clause in method signatures
- Explore the role of the throw clause in input validation, error propagation, and creating custom exceptions.

STRUCTURE:

- 11.1 Errors in Java Program
 - 11.1.1. Syntax Errors
 - 11.1.2. Runtime Errors
 - 11.1.3. Logical Errors:
 - 11.1.4. Compilation Errors
- 11.2 Exceptions
 - 11.2.1 Various Categories of Exceptions
 - 11.2.1.1. Checked Exceptions
 - 11.2.1.2. Runtime Exceptions
 - 11.2.1.3 Errors
- 11.3 Hierarchy of Exceptions
 - 11.3.1 Built in Exceptions
- 11.4 Exception Handling
 - 11.4.1 try – catch block
 - 11.4.1.1 Without Exception Handling
 - 11.4.1.2 With Exception Handling
 - 11.4.2 Multiple catch blocks
 - 11.4.3 Example: Using a Single Catch Block for Multiple Exceptions
 - 11.4.4 Finally block
 - 11.4.4.1 Syntax :
 - 11.4.4.2 How the 'finally' Block Works
- 11.5 Throws Clause
- 11.6 Throw Clause
- 11.7 Summary
- 11.8 Technical Terms
- 11.9 Self-Assessment Questions
- 11.10 Further Readings

11.1 ERRORS IN JAVA PROGRAM

Errors in Java can be broadly classified into several categories.

11.1.1. Syntax Errors:

Syntax errors are errors in the structure of our code, usually detected at compile time. These type of errors arise if rules of the language are not followed.

- Examples:
 - Missing semicolons (;).
 - Mismatched braces ({}, [], `()).
 - Incorrect method signatures.

- Example Code:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World" // Missing closing
parenthesis
    }
}
```

11.1.2. Runtime Errors:

Runtime errors are errors that occur while the program is running, leading to abnormal termination. These type of errors occur because the program tries to perform an operation that is impossible to complete.

- Examples:
 - Division by zero ('ArithmeticException').
 - Null pointer dereference ('NullPointerException').
 - Array index out of bounds ('ArrayIndexOutOfBoundsException').

- Example Code:

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = new int[5];
        System.out.println(numbers[10]); //
ArrayIndexOutOfBoundsException
    }
}
```

11.1.3. Logical Errors:

Logical errors are errors in the logic of your code that lead to incorrect results or behavior. Logical error indicates that logic used for coding doesn't produce expected output.

- Examples:
 - Incorrect algorithm implementation.
 - Misuse of conditional statements.

- Example Code:

```
java
public class Main {
    public static void main(String[] args) {
        int x = 10;
        if (x > 5) {
            System.out.println("x is less than 5"); // Incorrect
message
        }
    }
}
```

11.1.4. Compilation Errors:

Errors that prevent the code from being compiled into bytecode.

- Examples:
 - Missing imports.
 - Type mismatch.
- Example Code:

```
public class Main {
    public static void main(String[] args) {
        int num = "Hello"; // Type mismatch error
    }
}
```

11.2 EXCEPTIONS

An exception is an event that usually signals an erroneous situation at run time. In java, exceptions are wrapped up as objects and can be dealt in one of three ways:

- ignore it,
- handle it where it occurs
- handle it at an another place in the program.

The exception object stores information about the nature of the problem. For example, due to network problem or class not found etc.

A Java Exception is an object that describes the exception that occurs in a program. When an exceptional events occurs in java, an exception is said to be thrown. The code that's responsible for doing something about the exception is called an *exception handler*.

11.2.1 Various Categories of Exceptions

11.2.1.1. Checked Exceptions

The first type of exception is known as a checked exception, and it is an exception that is often caused by a user error or a problem that the programmer was unable to anticipate. Another way to define checked exceptions is as follows: "Checked exceptions are the classes that extend the Throwable class with the exception of Runtime Exception and Error." An example of an exception would be the situation in which a file is supposed to be opened but the file cannot be located. It is not possible to simply disregard these exceptions during the compilation process because they are examined throughout the compilation process. The IO Exception, SQL Exception, and other exceptions are examples of checked exceptions.

11.2.1.2. Runtime Exceptions

Exceptions that are not checked during compilation are referred to as runtime exceptions. These exceptions are ignored during the compilation process, but they are examined when the program is being executed. In addition, the term "Unchecked Exceptions" can be described as "The Classes that extend the Runtime Exception class are known as Unchecked Exceptions." Examples of exceptions include the Arithmetic Exception and the Null Pointer Exception.

11.2.1.3 Errors:

Errors are not exceptions at all; rather, they are problems that originate from circumstances that are beyond the control of either the user or the programmer. Errors are often disregarded in your code because it is quite unusual that you are able to take any action to correct a mistake. Errors will be generated, for instance, in the event that a stack overflow takes place. At the time of compilation, they are also disregarded as irrelevant.

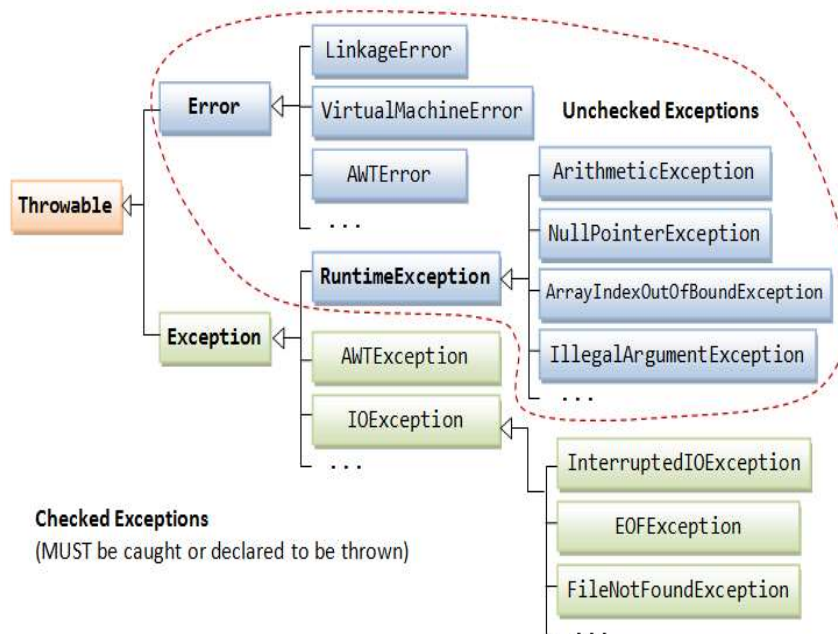


Figure 11.1 Types of Exceptions

Differences between checked and unchecked exceptions

Checked Exceptions	Unchecked Exceptions
<ul style="list-style-type: none"> • represent invalid conditions in areas outside the immediate control of the program • checked at compile time • The exception that can be predicted by the programmer • The classes that extend Throwable class except Runtime Exception and Error are known as checked exceptions • e.g. IO Exception, SQL Exception etc. Checked exceptions are checked at compile-time. 	<ul style="list-style-type: none"> • represent defects in the program (bugs) • checked at run time • Unchecked exception are ignored at compile time. • The classes that extend Runtime Exception are known as unchecked exceptions • e.g. Arithmetic Exception, Null Pointer Exception, Array Index Out Of Bounds Exception etc.

11.3 HIERARCHY OF EXCEPTIONS

All exception classes are subtypes of the `java.lang.Exception` class. The exception class is a subclass of the `Throwable` class. Other than the exception class there is another subclass called `Error` which is derived from the `Throwable` class

In Java, exceptions are organized in a hierarchy that extends from the base class `Throwable`. Understanding this hierarchy helps in properly handling exceptions and debugging issues in your code.

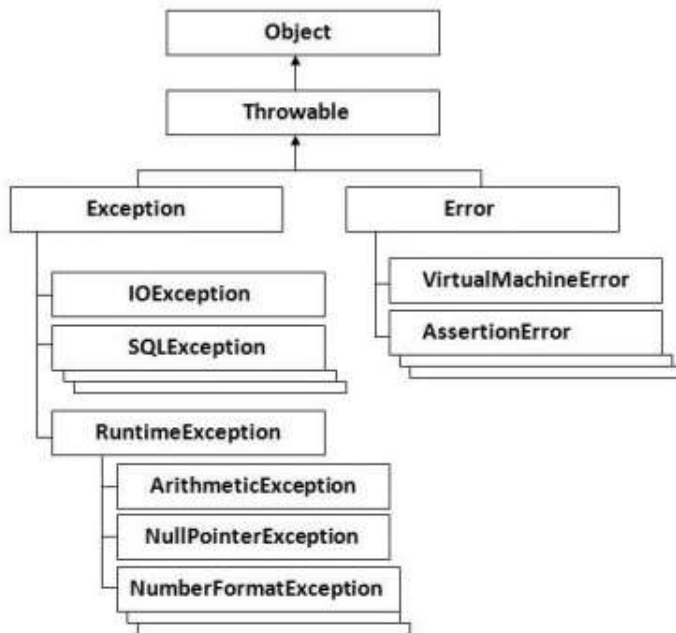


Figure 11.2 Exception hierarchy

11.3.1 Built in Exceptions:

List of Java Unchecked exceptions under Runtime Exception

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

List of Java Checked Exceptions defined in java.lang

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

11.4 EXCEPTION HANDLING

Exception Handling is the mechanism to handle runtime malfunctions. We need to handle such exceptions to prevent abrupt termination of program. The term exception means exceptional condition, it is a problem that may arise during the execution of program. A bunch of things can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc

The responsibility of Exceptional Handling is in charge of ensuring that the program continues to flow normally. In order to accomplish this, we first make an effort to capture the exception object that is thrown by the incorrect condition, and then we should display the proper message so that our actions can be corrected.

keywords are used to handle exceptions in Java

1. try
2. catch
3. finally
4. throw
5. throws

11.4.1 try – catch block

A method captures an exception by employing a combination of the try and catch operations. A try/catch block is employed around the code that has the potential to produce an exception. Code included within a try/catch block is known as protected code

Syntax:

```
try
{
    //Protected Code
} catch(ExceptionName e1)
{
    //Catch block
}
```

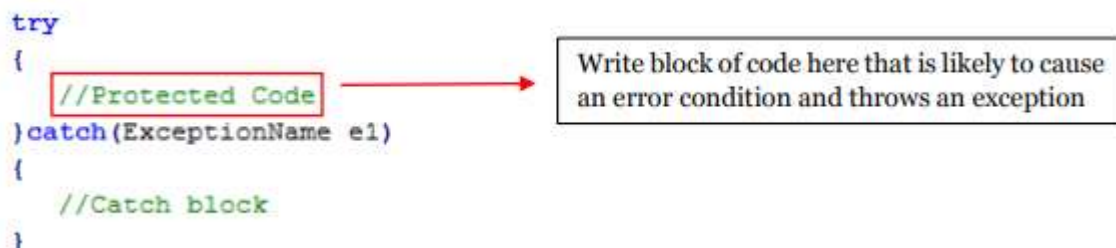
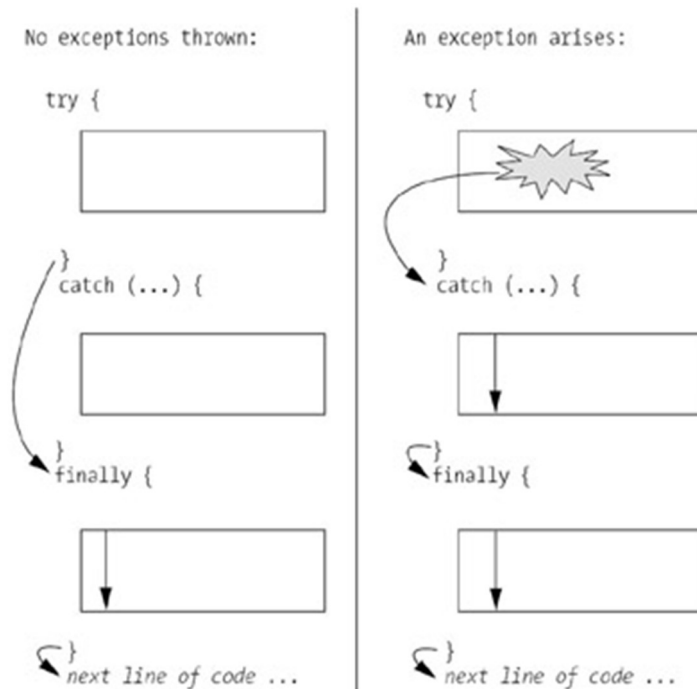


Figure 11.3 try catch block

An exception catch statement is the declaration of the specific type of exception that is being attempted to catch. Whenever an exception arises in protected code, the catch block (or blocks) that immediately follows the try statement is examined. If the specific sort of exception that has taken place is specified in a catch block, the exception is transferred to the catch block in a similar manner as an argument is transferred to a method parameter.

Let's look at the below two Java programs that demonstrate dividing by zero, one without exception handling and one with exception handling.



11.4.1.1 Without Exception Handling

This program demonstrates what happens when you attempt to divide by zero without handling the exception. In Java, dividing an integer by zero will throw an `ArithmeticException`.

```
public class DivideByZeroWithoutHandling {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;

        // Attempting to divide by zero
        int result = numerator / denominator; // This line will throw an
        ArithmeticException

        System.out.println("Result: " + result); // This line will not be
        executed
    }
}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

at DivideByZeroWithoutHandling.main(DivideByZeroWithoutHandling.java:7)

As seen from the output, the program terminates abruptly with an `ArithmeticException`.

Without Exception Handling, the program throws an `ArithmeticException` and terminates immediately when the exception occurs.

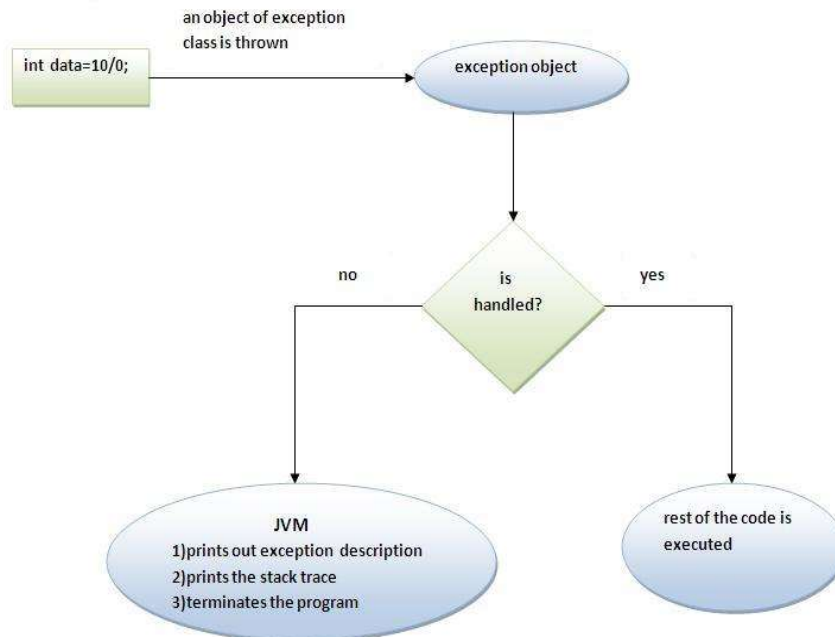


Figure 11.4 without exception handling

11.4.1.2 With Exception Handling

This program shows how to handle a divide-by-zero exception using a try-catch block. It allows the program to continue running even if an exception occurs.

```

public class DivideByZeroWithHandling {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;

        try {
            // Attempting to divide by zero
            int result = numerator / denominator;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            // Handling the exception
            System.out.println("Error: Cannot divide by zero.");
        }

        System.out.println("Program continues after handling the exception.");
    }
}
  
```

Output:

```

Error: Cannot divide by zero.
Program continues after handling the exception.
  
```

With Exception Handling, the exception is caught in the catch block, allowing the program to print a user-friendly error message and continue executing the rest of the code.

Using exception handling is crucial in ensuring that your program can handle errors gracefully and continue running or terminate in a controlled manner.

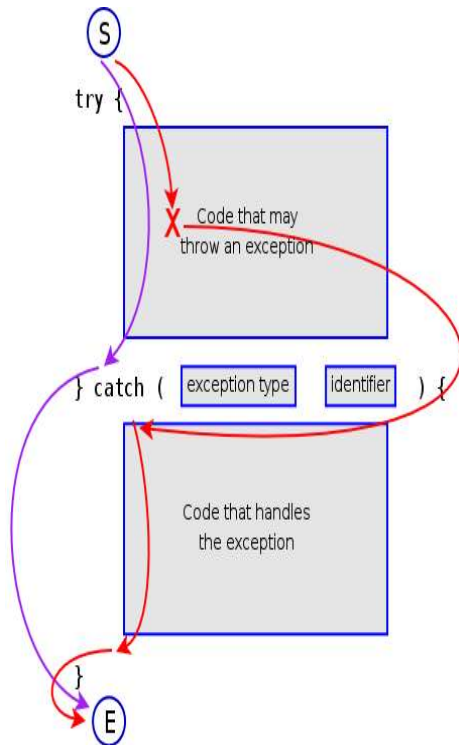


Fig 11.5 try-catch block

11.4.2 Multiple catch blocks

In Java, multiple catch blocks allow you to handle different types of exceptions that might be thrown by a 'try' block. Each catch block is used to handle a specific type of exception. This helps in writing more precise and specific error-handling code, enabling the program to respond differently depending on the type of exception that occurs.

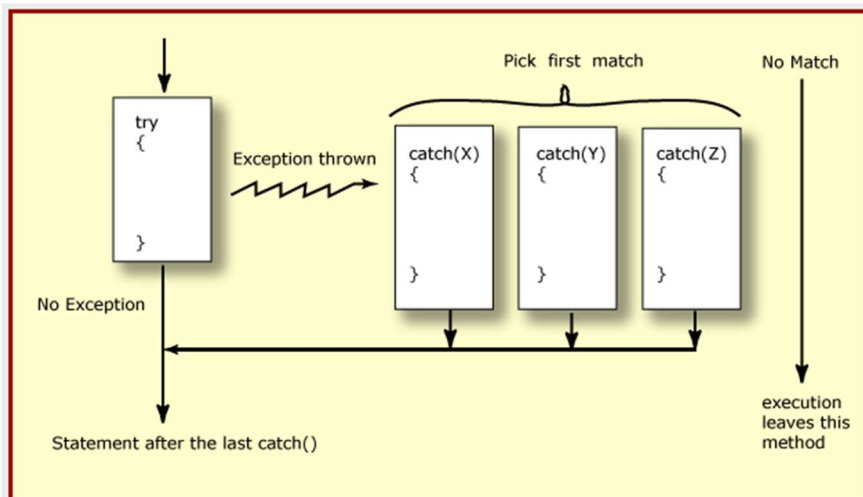


Figure 11.6 multiple catch blocks

Example:

```
public class MultipleCatchBlocks {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
        }
    }
}
```

```
        int result = 10 / 0; // This will throw an ArithmeticException
        System.out.println(numbers[3]); // This will throw an
        ArrayIndexOutOfBoundsException
    } catch (ArithmeticException e) {
        System.out.println("Caught an ArithmeticException: " +
e.getMessage());
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Caught an ArrayIndexOutOfBoundsException: "
+ e.getMessage());
    } catch (Exception e) {
        System.out.println("Caught a general exception: " +
e.getMessage());
    }

    System.out.println("Program continues after handling exceptions.");
}
}
```

Output:

Caught an ArithmeticException: / by zero
Program continues after handling exceptions.

In the above program

'try' Block Contains code that may throw different types of exceptions. In this example:

- 'int result = 10 / 0;' throws an 'ArithmeticException'.
- 'System.out.println(numbers[3]);' would throw an 'ArrayIndexOutOfBoundsException', but it is never reached because of the previous exception.

Catch Blocks 'catch (ArithmeticException e)': This block catches 'ArithmeticException' and handles it by printing a message. Since the exception is caught here, the other catch blocks are not executed.

- 'catch (ArrayIndexOutOfBoundsException e)': This block would catch an 'ArrayIndexOutOfBoundsException' if it occurred.
- 'catch (Exception e)': This is a general catch block that catches any exception that is not caught by the previous blocks. It acts as a fallback.

The important things we have to consider in exception handling mechanism are :

- Order of Catch Blocks: Catch blocks should be ordered from the most specific to the most general. This is because Java checks each catch block in sequence, and the first block that matches the exception type will be executed. If a more general exception type (like 'Exception') is caught before a more specific one (like 'ArithmeticException'), the specific block will never be reached, which can lead to compilation errors.
- Handling Multiple Exceptions: You can handle multiple exceptions of different types in the same try block by defining multiple catch blocks. This makes your code more robust and easier to debug.
- Common Superclass: If you want to handle exceptions that share a common superclass, you can catch the superclass type. For example, catching 'Exception' will catch any checked or unchecked exceptions.

11.4.3 Example: Using a Single Catch Block for Multiple Exceptions

Java 7 introduced multi-catch blocks, allowing multiple exceptions to be caught in a single catch block using the '|' (pipe) symbol.

```
public class MultiCatchExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            int result = 10 / 0; // This will throw an ArithmeticException
            System.out.println(numbers[3]); // This will throw an
            ArrayIndexOutOfBoundsException
        } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught an exception: " + e.getMessage());
        }

        System.out.println("Program continues after handling exceptions.");
    }
}
```

Output:

Caught an exception: / by zero

Program continues after handling exceptions.

This code is shorter and less repetitive when multiple exceptions are handled in the same way. However, if you need different handling logic for different exceptions, using separate catch blocks is still the preferred approach.

11.4.4 Finally block

In Java, the 'finally' block is a block of code that is always executed after the 'try' block, regardless of whether an exception was thrown or caught. It is typically used to perform clean-up operations, such as closing files, releasing resources, or resetting variables, ensuring that these actions are always executed no matter what happens in the 'try' block.

11.4.4.1 Syntax :

The 'finally' block is written after the 'try' and any associated 'catch' blocks. It is optional but is often used when resources need to be cleaned up.

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e1) {
    // Handle exception of type ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle exception of type ExceptionType2
} finally {
    // Code that will always be executed after the try and catch blocks
}
```

11.4.4.2 How the 'finally' Block Works

- Always Executed: The code inside the 'finally' block is always executed, even if an exception is thrown and caught, or even if there is a return statement in the 'try' or 'catch' blocks.
- Use Cases: It is used for code that must execute regardless of whether an exception occurs or not, such as closing files or network connections, releasing locks, or cleaning up memory.

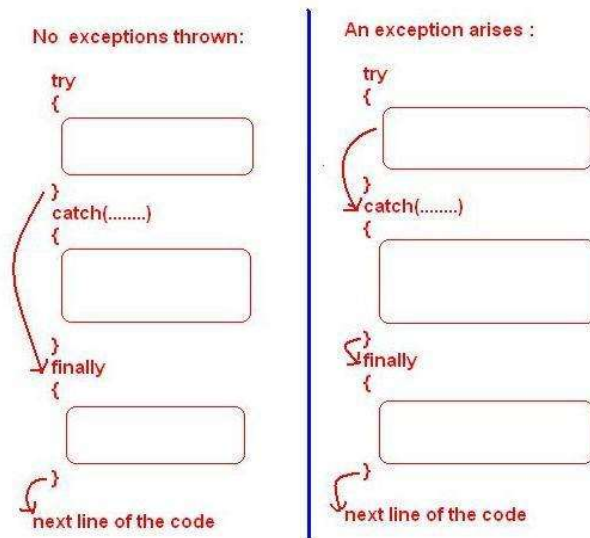


Figure 11.7 Finally block

Example

```
public class FinallyBlockExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);    // This will throw an
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Caught an ArithmeticException: " +
            e.getMessage());
        } finally {
            System.out.println("This is the finally block. It always
            executes.");
        }

        System.out.println("Program continues after the try-catch-finally
        block.");
    }

    public static int divide(int a, int b) {
        return a / b;
    }
}
```

Output:

Caught an ArithmeticException: / by zero
 This is the finally block. It always executes.
 Program continues after the try-catch-finally block.

The things we have to remember during the usage of finally keyword are:

- **Always Executes:** The 'finally' block executes regardless of whether an exception is thrown or caught in the 'try' or 'catch' blocks.
- **Resource Management:** It is ideal for closing resources such as file streams, database connections, and sockets. This ensures that resources are properly released even if an exception occurs.

- **Exception in 'finally' Block:** If an exception is thrown inside the 'finally' block, it will override any exception thrown in the 'try' or 'catch' blocks. It's generally advisable to avoid throwing exceptions from the 'finally' block to prevent losing the original exception.
- **Return Statements:** If there are return statements in the 'try' or 'catch' blocks, the 'finally' block will still execute. However, if there is a return statement in the 'finally' block, it will override any previous return values from the 'try' or 'catch' blocks.

11.5 THROWS CLAUSE

In Java, the 'throws' clause is used in a method declaration to specify that the method might throw one or more exceptions. It informs the compiler and developers using the method that they need to handle these exceptions. The 'throws' clause is typically used with checked exceptions (exceptions that are checked at compile time).

Syntax

The 'throws' clause is added to the method signature and lists the exceptions that the method may throw. If a method does not handle a checked exception (i.e., does not use a 'try-catch' block), it must declare it using the 'throws' clause.

```
public void methodName() throws ExceptionType1, ExceptionType2 {  
    // Method code that might throw ExceptionType1 or ExceptionType2  
}
```

Example:

```
import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
  
public class ThrowsExample {  
    public static void main(String[] args) {  
        try {  
            readFile("example.txt");  
        } catch (IOException e) {  
            System.out.println("Caught IOException: " + e.getMessage());  
        }  
    }  
  
    // Method declaring that it might throw IOException  
    public static void readFile(String fileName) throws IOException {  
        BufferedReader reader = new BufferedReader(new  
FileReader(fileName));  
        String line = reader.readLine();  
        System.out.println(line);  
        reader.close();  
    }  
}
```

The above program illustrates the following points:

1. 'throws IOException' in 'readFile' Method:

- The 'readFile' method declares that it might throw an 'IOException' using the 'throws' clause.
- The method performs file I/O operations, which might result in an 'IOException' if the file does not exist or cannot be read.

2. Handling the Exception:

- The 'main' method calls 'readFile' within a 'try' block and catches the potential 'IOException' using a 'catch' block.
- This way, 'main' handles the exception, preventing the program from crashing.

We can also observe the following things from the above program:

1. **Checked Exceptions:** The 'throws' clause is mainly used for checked exceptions, which must be handled either by a 'try-catch' block or by declaring them in the method signature using 'throws'.
2. **Unchecked Exceptions:** Unchecked exceptions (subclasses of 'RuntimeException') do not need to be declared or handled. Therefore, the 'throws' clause is generally not used for them, although it can be if desired for clarity.
3. **Method Signature:** When a method declares a 'throws' clause, it becomes part of the method signature. Any code calling this method must handle the exceptions listed in the 'throws' clause, either by catching them or by declaring them in its own 'throws' clause.
4. **Multiple Exceptions:** A method can declare multiple exceptions in the 'throws' clause, separated by commas.

11.6 THROW CLAUSE

In Java, the 'throw' clause is used to explicitly throw an exception from a method or any block of code. The 'throw' statement allows you to create an exception and then pass it to the runtime system, which searches for an appropriate 'catch' block to handle the exception.

Syntax :

```
throw new ExceptionType("Error message");
```

Here, 'ExceptionType' is the type of the exception you want to throw (such as 'ArithmeticException', 'NullPointerException', 'IOException', etc.), and "Error message" is a string that provides additional details about the exception.

Example

```
public class ThrowExample {
    public static void main(String[] args) {
        try {
            checkNumber(-5);
        } catch (IllegalArgumentException e) {
            System.out.println("Caught an exception: " + e.getMessage());
        }
    }

    public static void checkNumber(int number) {
        if (number < 0) {
            throw new IllegalArgumentException("Number must be non-
negative"); // Throwing an exception
        }
        System.out.println("Number is: " + number);
    }
}
```

Output:

Caught an exception: Number must be non-negative

The above program illustrates the following things

1. 'throw new IllegalArgumentException("Number must be non-negative");': This line explicitly throws an 'IllegalArgumentException' if the 'number' is negative. The message "Number must be non-negative" is passed to the exception, providing details about what went wrong.
2. Catching the Exception: In the 'main' method, the 'checkNumber' method is called within a 'try' block. If the 'checkNumber' method throws an exception, it is caught by the 'catch' block, which prints a message to the console.

When to Use the 'throw' Clause:

- Input Validation: To validate inputs or arguments passed to methods. If an argument does not meet the required criteria, an exception can be thrown to indicate an error.
- Custom Exceptions: When creating custom exceptions, the 'throw' clause is used to throw these exceptions. This can provide more specific error messages and help in debugging.
- Error Propagation: To propagate exceptions to higher levels of the program where they can be handled appropriately.

Example of Throwing a Custom Exception

```
// Custom exception class
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught a custom exception: " +
e.getMessage());
        }
    }

    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older to
register"); // Throwing a custom exception
        }
        System.out.println("Age is valid for registration.");
    }
}
```

Output:

Caught a custom exception: Age must be 18 or older to register

Using the 'throw' statement effectively allows for precise error handling and helps maintain robust and reliable code.

11.7 SUMMARY

Java exception handling is a mechanism that helps manage errors and exceptions, ensuring robust and error-free code execution. This chapter covers various types of errors in Java programs, such as syntax errors, runtime errors, and logical errors. It explores the hierarchy of exceptions, starting from the base class Throwable and branching into Error and Exception subclasses, with further distinctions between checked and unchecked exceptions.

The chapter discusses key concepts in exception handling, including the try-catch blocks for capturing and managing exceptions, the finally block for executing cleanup code, the throws clause for declaring exceptions that a method might throw, and the throw clause for explicitly throwing exceptions. Together, these tools allow developers to write more reliable and maintainable Java programs by properly handling unexpected events and errors

11.8 TECHNICAL TERMS

Error, exception, try, catch, throw, throws

11.9 SELF ASSESSMENT QUESTIONS

Essay questions:

1. What is exception handling in Java, and why is it important? explain with examples
2. Explain with examples the use of multiple catch blocks in handling different exceptions.
3. How do you explicitly throw an exception in Java? Provide an example.
4. Give an example of a method that uses the throws clause to indicate a checked exception.

Short Answer Questions:

1. What are the different types of errors in a Java program?
2. What is an Error in Java, and how is it different from an Exception?
3. Explain the difference between checked and unchecked exceptions in Java.
4. What happens if an exception is thrown in the finally block?

11.10 SUGGESTED READINGS

- 1) Herbert Schildt and Dale Skrien “Java Fundamentals –A comprehensive Introduction”, McGraw Hill, 1st Edition, 2013.
- 2) Herbert Schildt, “Java the complete reference”, McGraw Hill, Osborne, 11th Edition, 2018.
- 3) T. Budd “Understanding Object-Oriented Programming with Java”, Pearson Education, Updated Edition (New Java 2 Coverage), 1999 REFERENCE BOOKS:
- 4) P.J. Dietel and H.M. Dietel “Java How to program”, Prentice Hall, 6th Edition, 2005.
- 5) P. Radha Krishna “Object Oriented programming through Java”, CRC Press, 1st Edition, 2007.
- 6) Malhotra and S. Choudhary “Programming in Java”, Oxford University Press, 2nd Edition, 2014

LESSON- 12

THREADS

OBJECTIVES:

After going through this lesson, you will be able to

- Understand what single-tasking is and how it contrasts with multi-tasking
- Differentiate between process-based and thread-based multi-tasking
- Learn about the various uses of threads in Java
- Learn the different ways to create a thread in Java
- Familiarize with important methods of the Thread class

STRUCTURE:

- 12.1. Introduction
- 12.2. Single Tasking
- 12.3. Multi-Tasking
- 12.4. Uses of Threads
- 12.5. Thread Life Cycle
- 12.6. Creating a Thread and Running it
- 12.7. Terminating the Thread
 - 12.7.1 Using a Volatile Flag
 - 12.7.2. Using the interrupt () Method:
- 12.8. Thread Class Methods
- 12.9. Summary
- 12.10. Technical Term
- 12.11. Self-Assessment Question
- 12.12. Further Readings

12.1 INTRODUCTION

The direction or path that is taken while a program is being executed is referred to as a thread in the Java programming language. In general, every program has at least one thread, which is referred to as the main thread. This thread is provided by the Java Virtual Machine (JVM) at the beginning of the execution of the program. At this moment, the main thread is the one that calls the main() method. This occurs when the main thread is specified.

The execution thread of a program is referred to as a thread. When an application is executing on the Java Virtual Machine, it is possible for the application to run many threads of execution simultaneously. Some threads have a higher priority than others. The execution of higher priority threads comes before the execution of lower priority threads.

The reason why thread is so important to the program is that it makes it possible for several actions to take place within a single procedure. In many cases, the program counter, stack, and local variable are all assigned to each individual thread in the program.

Java's Thread feature allows for concurrent execution, which helps to divide work and increase overall speed. When it comes to successfully managing processes such as

input/output and network connection, it is absolutely necessary. For Java applications to be responsive, having a solid understanding of threads is essential.

12.2 SINGLE TASKING

Single-tasking refers to a mode of operation in computing where only one task or process is executed at a time. In a single-tasking system, the CPU handles one task and then moves on to the next only after the current task has been completed. Executing the tasks is of two types. One is single tasking and another is multi-tasking

Single Tasking is executing only one task at a time is called single tasking. In this single tasking the microprocessor will be sitting idle for most of the time. This means microprocessor time is wasted.

Characteristics of Single-Tasking:

- Sequential Execution: Only one process or application is active at any given moment. Once the task finishes, another can start.
- Resource Utilization: System resources (CPU, memory) are dedicated to a single task, which can simplify resource management but limit overall system efficiency.
- Simplicity: Single-tasking systems are simpler to implement because there's no need to manage multiple tasks simultaneously.

Example:

An old-style operating system like MS-DOS is an example of a single-tasking environment. When running MS-DOS, you could only execute one program at a time. Once you finished running a program, you could start another one.

12.3 MULTI TASKING

The term "multitasking" refers to the capability of a computer system to carry out many tasks at the same time or in time intervals that overlap with one another. It is possible to accomplish this through two different methods: multitasking by utilizing several processes or multitasking by utilizing numerous threads. The utilization of threads is the primary emphasis of Java's multitasking capabilities.

There exist two clearly identified categories of multitasking:

- Process-based and
- Thread-based.

It is crucial to clarify the distinction between two. Process is the term used to define the Program in active execution. Thus, process-based multitasking is the capability that enables your computer to execute two or more programs simultaneously. For instance, we can concurrently utilize the Java compiler and text editor services. One other illustration is our capacity to perceive the music and simultaneously obtain the printed materials from the printer.

The thread is the fundamental unit of dispatchable code in the thread-based multitasking environment. This implies that a single program has the capability to include multiple components, each of which is referred to as a Thread module. For instance, the text

editor has the capability to both Format the text and Print it. While Java applications utilize process-based multitasking environments, the specific architecture of these environments is not well defined.

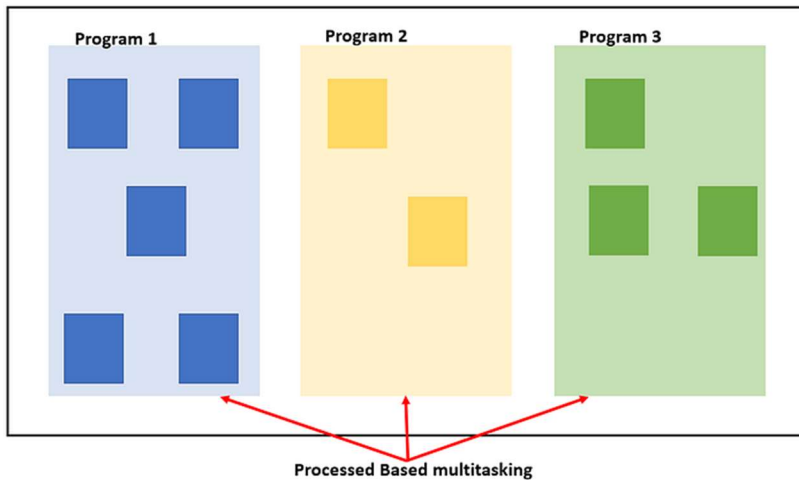


Figure 12.1: Process based multitasking

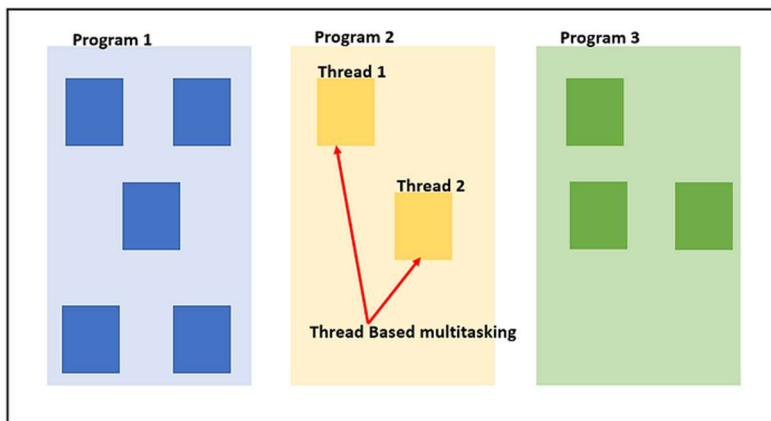


Figure 12.2: Thread based multitasking

Process-Based Multitasking	Thread – based Multitasking
<ul style="list-style-type: none"> • This deals with "Big Picture" • These are Heavyweight tasks • Inter-process communication is expensive and limited • Context switching from one process to another is costly in terms of memory • This is not under the control of Java 	<ul style="list-style-type: none"> • This deals with Details • These are Lightweight tasks • Inter-Thread communication is inexpensive. • Context switching is low cost in terms of memory, because they run on the same address space • This is controlled by java

12.4 USES OF THREADS

Threads in Java are used to achieve concurrent execution, which allows multiple tasks to be performed simultaneously or in overlapping periods. This can improve the performance and responsiveness of applications.

1. Improving Application Responsiveness

- **User Interfaces:** In GUI applications (e.g., Swing, JavaFX), threads are used to keep the user interface responsive. For example, background tasks like loading data or performing computations are run in separate threads so that the main UI thread remains responsive to user inputs.

2. Parallel Processing

- **Data Processing:** Threads can be used to process large datasets in parallel. For instance, you can split data into chunks and process each chunk in a separate thread to speed up computation.

3. Asynchronous Operations

- **Non-blocking Tasks:** Threads allow for asynchronous execution of tasks that would otherwise block the main thread, such as network calls or file I/O operations. This ensures that other tasks can proceed while waiting for the asynchronous operation to complete.

4. Handling Multiple Client Connections

- **Server Applications:** In server applications, such as web servers or chat servers, threads are used to handle multiple client connections simultaneously. Each client request can be processed in a separate thread, allowing the server to handle multiple requests concurrently.

5. Real-Time Systems

- **Real-Time Processing:** In systems that require real-time processing, such as video games or real-time data analysis, threads can be used to ensure that tasks are performed within strict timing constraints.

6. Periodic Tasks

- **Scheduled Tasks:** Threads are used to perform periodic tasks, such as regular data updates or scheduled maintenance tasks. The 'ScheduledExecutorService' can be used to schedule tasks to run at fixed intervals or after a delay.

7. Background Tasks

- **Long-Running Operations:** Threads are useful for executing long-running background tasks without blocking the main execution flow. For example, you might use threads to perform background data processing or resource loading.

8. Parallel Algorithms

- **Computational Algorithms:** Many algorithms can benefit from parallel execution. Threads can be used to implement parallel algorithms, such as divide-and-conquer strategies or parallel sorting algorithms.

9. Task Coordination

- **Coordinating Tasks:** Threads can be used to coordinate complex task execution flows, such as task dependencies and inter-task communication. Java provides mechanisms like 'CountDownLatch', 'CyclicBarrier', and 'Semaphore' to manage coordination between threads.

10. Thread Pools and Resource Management

- **Efficient Resource Use:** Thread pools are used to efficiently manage a large number of tasks by reusing a fixed number of threads. This avoids the overhead of creating and destroying threads frequently and helps in managing system resources.

Threads in Java provide a powerful mechanism for concurrent and parallel processing, allowing for improved performance, responsiveness, and resource management. They enable various use cases from improving application responsiveness and handling multiple client connections to performing parallel computations and managing periodic tasks. Properly managing threads and ensuring thread safety are crucial for building robust and efficient multi-threaded applications.

12.5 THREAD LIFE CYCLE

The Java thread lifecycle refers to the various stages that a thread undergoes from its creation to its termination.

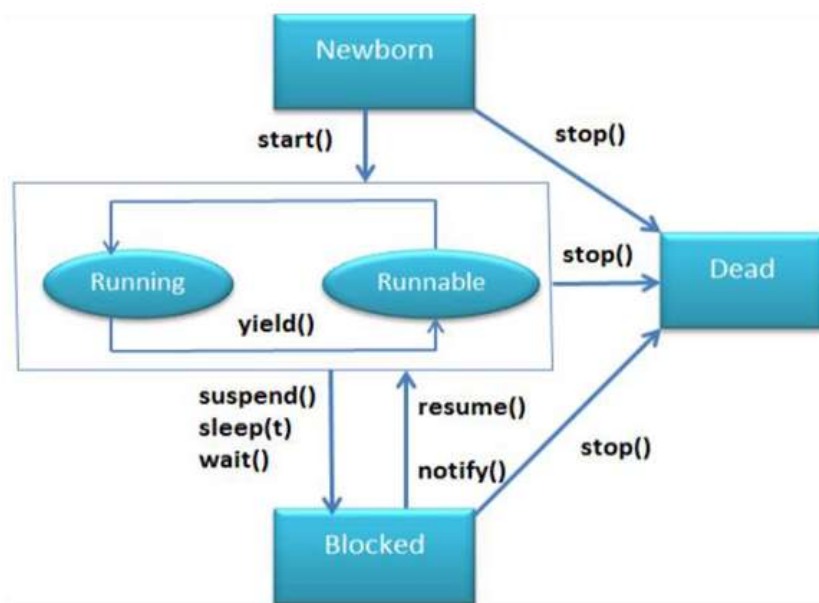


Figure 12.3 Thread life cycle

- **New:** When a thread is first created but hasn't yet started running, it is in the "New" state. At this point, the thread is not eligible for execution.
- **Runnable:** Once the thread's `start()` method is called, the thread moves to the "Runnable" state. This doesn't mean the thread is currently running; it simply means it's ready to run and is waiting for CPU time. The thread could be actively running or just waiting for its turn to execute.
- **Blocked:** A thread enters the "Blocked" state when it is waiting to acquire a lock or resource that is currently held by another thread. It will remain blocked until the resource becomes available.
- **Waiting:** In the "Waiting" state, a thread is waiting indefinitely for another thread to perform a particular action. This state is typically reached by calling methods like `Object.wait()`.
- **Timed Waiting:** This is similar to the "Waiting" state, but the thread waits for a specific period of time. It might use methods like `Thread.sleep()` or `Object.wait(long timeout)` to enter this state. The thread will return to the runnable state either when the specified time elapses or when another thread interrupts it.
- **Terminated:** A thread moves to the "Terminated" state when it has finished its execution. This means the thread has completed its `run()` method or has been terminated due to an exception.

Below is the execution flow of Thread life cycle i.e. how a thread goes into ready state from born state and from ready to running and finally into Dead state.

Initially, a thread is in the "New" state after being instantiated but not yet started. Once the `start()` method is invoked, the thread transitions to the "Runnable" state, where it is eligible for execution by the CPU. During its execution, a thread may enter the "Blocked" state if it needs to wait for a resource or lock held by another thread, or the "Waiting" state if it waits indefinitely for a specific condition. It can also be in the "Timed Waiting" state if it waits for a specific period. Finally, a thread moves to the "Terminated" state upon completing its task or if it is terminated due to an exception. Understanding this lifecycle is crucial for effective thread management and synchronization in Java applications.

12.6 CREATING A THREAD AND RUNNING IT

In Java, there are two primary ways to create a thread:

1. Extending the 'Thread' Class:

- We can create a new thread by subclassing the 'Thread' class and overriding its 'run()' method. This method contains the code that will be executed when the thread starts. After creating an instance of your subclass, we invoke the 'start()' method to begin execution.

```
class MyThread extends Thread {
    public void run() {
        // Code to be executed by the thread
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

2. Implementing the 'Runnable' Interface:

- Another way to create a thread is by implementing the 'Runnable' interface, which requires we to define the 'run()' method. You then pass an instance of our 'Runnable' implementation to a 'Thread' object and start the thread by calling its 'start()' method.

```
class MyRunnable implements Runnable {
    public void run() {
        // Code to be executed by the thread
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t = new Thread(myRunnable);
        t.start();
    }
}
```

Both approaches have their use cases. Extending 'Thread' is straightforward but limits your ability to inherit from other classes since Java supports single inheritance. Implementing 'Runnable' is more flexible, allowing our class to extend other classes while still being able to run in its own thread.

12.7 TERMINATING THE THREAD

A terminated thread means it is dead and no longer available.

A thread may remain in the terminated state for the following reasons:

- Termination occurs when a thread normally finishes its work.
- Sometimes threads may terminate due to unusual events like segmentation faults, exceptions. Such termination may be termed abnormal termination.

Terminating a thread in Java requires a cooperative approach.

12.7.1 Using a Volatile Flag:

Create a volatile boolean variable (e.g., `isRunning`) in your thread class.

In our thread's `run ()` method, periodically check the value of this flag. If it's set to false, exit the loop and terminate the thread.

From another thread, set the flag to false when you want to terminate the thread.

```
public class MyThread extends Thread {
    private volatile boolean isRunning = true;

    public void run() {
        while (isRunning) {
            // Do some work
        }
    }
    public void stopRunning() {
        isRunning = false;
    }
}
```

12.7.2. Using the `interrupt ()` Method:

Use the `interrupt ()` method on the thread you want to terminate.

Inside the thread's `run ()` method, check for the interrupt status using `Thread.interrupted ()` or catch the `InterruptedException`.

If the thread is interrupted, perform any necessary cleanup and exit the loop.

```
public class MyThread extends Thread {
    public void run() {
        try {
            while (!Thread.interrupted()) {
                // Do some work
            }
        } catch (InterruptedException e) {
            // Thread interrupted, perform cleanup
        }
    }
}
```

12.8 THREAD CLASS METHODS

The `Thread` class in Java provides several methods to manage thread behavior and interact with thread execution. The following picture depicts the various thread methods.

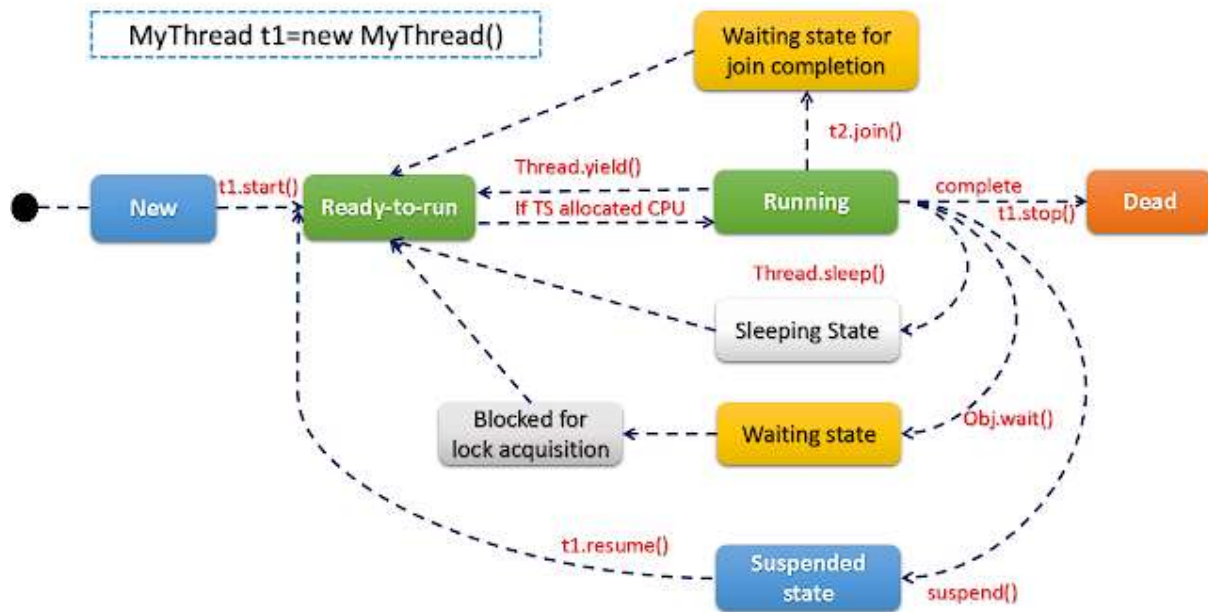


Figure 12.4: Thread class methods

When we write `MyThread t1=new MyThread()` then thread is in the New/Born state. When we call `t.start()` method then thread enters into Ready State or Runnable State. If Thread Scheduler allocates the processor to Thread then Thread enters into Running State. If `run()` method completes successfully then thread enters into Dead State. above are the basic main states of the Thread. but apart from this it has some condition through that it goes into different states like waiting state,suspended state,sleeping state.the full description is below.

start()

Begins the execution of the thread. It invokes the `run()` method in a new thread of execution.

```
Thread t = new Thread();
t.start();
```

run()

contains the code that constitutes the new thread's task. This method should be overridden in a subclass of `Thread` or in a `Runnable` object.

```
public void run() {
    // Thread code here
}
```

interrupt():

Interrupts the thread, setting its interrupt flag. If the thread is blocked in a method like `sleep()` or `wait()`, it will throw an `InterruptedException`.

```
thread.interrupt();
```


yield()

If a running thread calls the *Thread.yield()* method then thread enters into ready state from running state to give chance to other waiting thread of same priority immediately.

```
Thread.yield();
```

join()

If a Thread calls the *join()* method then it enters into waiting state and if this thread comes out from waiting state/blocked state then it enters into Ready/Runnable state but here is some condition to come out from the waiting state is-

A) If thread completes its own execution

B) If time expires.

C) If waiting thread got interrupted.

```
thread.join(); // Waits indefinitely for the thread to finish  
thread.join(1000); // Waits up to 1 second
```

sleep()

If running thread calls the *sleep()* then immediately enters into sleeping state. now thread will come out of this state to ready state only when-

A) If time expires.

B) If sleeping thread got interrupted.

This method can throw an *InterruptedException*.

```
Thread.sleep(1000); // Sleeps for 1 second
```

wait()

If thread calls the *wait()* method then running thread will enters into waiting state. if this thread got any notification by method *notify()/notifyAll()* then it enters into another waiting state to get lock.so when the thread comes out of waiting state to another waiting state to get lock is-

A) If waiting thread got notification.

B) If time expires.

C) If waiting thread got interrupted.

Now thread which is in the another waiting state will go to ready state when it get the lock.

```
synchronized (someObject) {  
    while (!condition) {  
        someObject.wait(); // Wait until notified or interrupted  
    }  
}
```

suspend()

If running thread calls the *suspend()* method now thread enters into suspended state and it will comes out form there to ready state only when it will call the *resume()* method.

```
thread.suspend(); // Deprecated method  
thread.resume(); // Deprecated method to resume
```

stop()

If running thread calls the *stop()* method then immediately enters into dead state.

```
thread.stop(); // Deprecated method
```

12.9 SUMMARY

The chapter on Java Threads explores the concept of threads and their importance in modern programming. It begins by contrasting single-tasking, where a single task is executed sequentially, with multi-tasking, which allows multiple tasks to run concurrently, enhancing the performance and responsiveness of applications. Threads are lightweight processes that facilitate multi-tasking in Java by enabling concurrent execution within a program. The chapter explains various uses of threads, such as performing background operations, improving application responsiveness, and managing multiple tasks simultaneously. It covers the two primary ways to create and run threads in Java: by extending the `Thread` class or implementing the `Runnable` interface. It also discusses how to manage the lifecycle of a thread, including starting and terminating threads, and provides an overview of key thread class methods, such as `start()`, `run()`, `sleep()`, and `join()`, which are essential for thread management and synchronization.

12.10 TECHNICAL TERMS

Thread, single tasking, multi-tasking, lightweight process, Synchronization.

12.11 SELF ASSESSMENT QUESTIONS

Essay questions:

1. What is a thread in Java? Describe the life cycle of a thread with a diagram.
2. Describe two ways to create a thread in Java, including code examples.
3. What are the key methods provided by the `Thread` class in Java? Describe at least five methods with examples.
4. How can a thread be terminated in Java? Discuss different ways to stop a thread, including the use of the `interrupt()` method.

Short Answer Questions:

1. How does multi-threading differ from multi-tasking?
2. Name two ways to create a thread in Java.
3. What is a daemon thread in Java?
4. Name the method used to check if a thread is alive.
5. What is the purpose of the `join()` method in the `Thread` class?

12.12 SUGGESTED READINGS

- 1) Herbert Schildt and Dale Skrien “Java Fundamentals –A comprehensive Introduction”, McGraw Hill, 1st Edition, 2013.
- 2) Herbert Schildt, “Java the complete reference”, McGraw Hill, Osborne, 11th Edition, 2018.
- 3) T. Budd “Understanding Object-Oriented Programming with Java”, Pearson Education, Updated Edition (New Java 2 Coverage), 1999 REFERENCE BOOKS:
- 4) P.J. Dietel and H.M. Dietel “Java How to program”, Prentice Hall, 6th Edition, 2005.
- 5) P. Radha Krishna “Object Oriented programming through Java”, CRC Press, 1st Edition, 2007.
- 6) Malhotra and S. Choudhary “Programming in Java”, Oxford University Press, 2nd Edition, 2014

Dr. U. SURYA KAMESWARI