OBJECT ORIENTED PROGRAMMING WITH JAVA
M.Sc (COMPUTER SCIENCE)
SEMESTER-I, PAPER-II

Lesson Writers:

Dr. K. Lavanya
Asst. Professor,
Dept. Of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510.

Dr.U. Surya Kameswari
Asst. Professor
Dept. Of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510.

Dr. Vasantha Rudramalla
Faculty
Dept. CS &E
Acharya Nagarjuna University,
Nagarjunanagar – 522 510

Mrs. Appikatla Pushpa Latha
Faculty, Dept. of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510

Editor

Dr.U. Surya Kameswari
Asst. Professor
Dept. of CS&E
Acharya Nagarjuna University
Nagarjunanagar – 522 510.

Director, I/c.

Prof. V. Venkateswarlu

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

Professor
Centre for Distance Education
Acharya Nagarjuna University
Nagarjuna Nagar 522 510

Ph: 0863-2346222, 2346208
0863- 2346259 (Study Material)
Website www.anucde.info
E-mail: anucdedirector@gmail.com

M.Sc Computer Science

First Edition : 2025

No. of Copies :

© Acharya Nagarjuna University

This book is exclusively prepared for the use of students of M.Sc (Computer Science), 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 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 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.Sc., B.A., B.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 endeavors.

Prof. K. Gangadhara Rao
M.Tech., Ph.D.,
Vice-Chancellor I/c
Acharya Nagarjuna University

M.Sc. Computer Science
Semester-I, Paper-II
102CP24 -OBJECT ORIENTED PROGRAMMING WITH JAVA

Syllabus

UNIT-1

Java Basics - History of Java, Java buzzwords, comments, data types, variables, constants, scop and life time of variables, operators, operator hierarchy, expressions, type conversion and casting, enumerated types, control flow-block scope, conditional statements, loops, break and continue statements, simple java program, arrays, input and output, formatting output, Review of OOP concepts, encapsulation, inheritance, polymorphism, classes, objects, constructors methods, parameter passing, static fields and methods, access control, this reference, overloading methods and constructors, recursion, garbage collection, building strings, exploring string class Enumerations, autoboxing and unboxing, Generics.

Inheritance -Inheritance concept, benefits of inheritance, Super classes and Sub classes, Member access rules, Inheritance hierarchies, super uses, preventing inheritance final classes and methods, casting, polymorphism- dynamic binding, method overriding, abstract classes and methods, the Object class and its methods.

UNIT II

Interfaces Interfaces vs. Abstract classes, defining an interface, implementing interfaces, accessing implementations through interface references, extending interface

Packages-Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages.

UNIT III

Files-streams- byte streams, character streams, text Input/output, binary input/output, random access file operations, File management using File class, Using java.io.

Exception handling - Dealing with errors, benefits of exception handling, the classification of exceptions- exception hierarchy, checked exceptions and unchecked exceptions, usage of try, catch, throw, throws and finally, rethrowing exceptions, exception specification, built in exceptions, creating own exception sub classes, Guide lines for proper use of exceptions.

UNIT IV

Multithreading - Differences between multiple processes and multiple threads, thread states, creating threads, interrupting threads, thread priorities, synchronizing threads, interthread communication, thread groups, daemon threads.

Event Handling Events, Event sources, Event classes, Event Listeners, Relationship between Event sources and Listeners, Delegation event model, Semantic and Low-level events, Examples handling a button click, handling mouse and keyboard events, Adapter classes.

UNIT V

Applets - Inheritance hierarchy for applets, differences between applets and applications, life cycle of an applet Four methods of an applet, Developing applets and testing, passing

(102CP24)

M.SC DEGREE EXAMINATION, Model QP

Computer Science – First Semester

OBJECT ORIENTED PROGRAMMING WITH JAVA

Time: 3 hrs **Max Marks: 70**

Answer ONE Question from each unit **5 x 14 = 70 M**

UNIT – I

1. a) Explain about final classes, final methods and final variables?

b) Explain about the abstract class with example program

(OR)

2. What are the basic principles of Object Oriented Programming? Explain with examples, how they are implemented in C++

UNIT – II

3. Is there any alternative solution for Inheritance. If so explain the advantages and disadvantages of it.

(OR)

4. a)What is a package? How do we design a package?

   b) How do we add a class or interface to a package?

UNIT – III

5. In JAVA, is exception handling implicit or explicit or both. Explain with the help of example java programs.

(OR)

6. a. Explain in detail about random access file operations.

   b. Write about Stream Classes.

UNIT-IV

7. a) With the help of an example, explain multithreading by extending thread class.

   b) Implementing Runnable interface and extending thread, which method you refer for multithreading and why?

(OR)

8. Explain Mouse and KeyBoard Events

UNIT – V

9. Differentiate following with suitable examples

   a) Frame, JFrame b) Applet, JApplet c) Menu, JMenu

(OR)

10. Explain the following

    a) Creating an applet b) Passing parameters to applets c) Adding graphics

CONTENTS

LESSON- 01

JAVA BASIC CONCEPTS

AIMS AND OBJECTIVES

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

- Know history of Java
- Explore Core Java Concepts.
- Understand the Real-time problem-solving with Java.
- Know Practical Application of Java.
- Learn some additional java concepts like casting and enumerated data types
- Understand the control flow of program

STRUCTURE

1.1 INTRODUCTION

Java is a widely used programming language known for its simplicity and versatility. Developed by Sun Microsystems in the mid-1990s, Java was designed to be platform-independent, allowing developers to write code that can run on any device equipped with the Java Virtual Machine (JVM). This introductory overview highlights Java's fundamental features, including a rich set of data types—both primitive and reference types—that help manage different kinds of information.

In Java, comments are crucial for documenting code, allowing developers to annotate their logic without impacting execution. Variables and constants are essential elements, where variables can change during execution, while constants remain fixed. Operators in Java facilitate various operations, such as arithmetic and logical calculations. Enumerated types provide a way to define a set of named values, enhancing code readability. The control flow structures, including conditionals (if-else statements) and loops (for and while), guide the execution path of the program. Importantly, block scope determines the visibility and lifetime of variables within defined blocks, ensuring organized and manageable code. Overall, Java's design promotes robust software development through its clear syntax and comprehensive features.

This chapter introduces Java, covers the fundamental features of java, history of Java, comment in java, data types, variables, constants, operators, type casting, enumerated types, and control flow-block scope

1.2 JAVA HISTORY

Java was developed by James Gosling and his team at Sun Microsystems in the mid-1990s, with its initial release occurring in 1995. The language was designed to address the shortcomings of C and C++, aiming for ease of use, security, and platform independence. Originally named "Oak," it was intended for programming set-top boxes for cable television. However, as the internet began to grow, Java's focus shifted towards web applications.

The release of Java 1.0 in 1995 marked a significant milestone, introducing features like the Java Virtual Machine (JVM) and the foundation for its "write once, run anywhere" capability. Subsequent versions brought enhancements, expanding the language's functionality and performance. Java 2 introduced the Collections framework and Swing for graphical user interface (GUI) development, while Java 5 (also known as J2SE 5.0) added generics and metadata.

Over the years, Java continued to evolve, with major versions such as Java 8 introducing lambda expressions and stream API, significantly enhancing its capabilities for modern programming paradigms. The transition to a time-based release model in 2017 allowed Java to adopt new features more rapidly. Today, Java remains a foundational language in

enterprise applications, Android development, and web services, celebrated for its stability, scalability, and extensive community support.

1.3 JAVA BUZZWORDS

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 1.1:

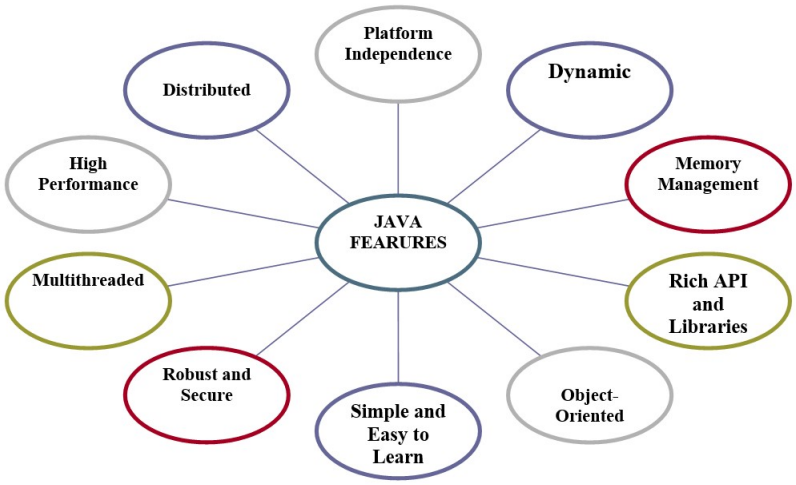


Figure 1.1 Java Buzzwords

❖ 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 and are shown in Table 1.1.

Table 1.1 Features of Java

Feature	Description
• Simple	• Java is easy to learn, and its syntax is quite simple, clean and easy to understand
• Object Oriented	• Java can be easily extended as it is based on Object Model
• Robust	• automatic Garbage Collector and Exception Handling.
• Platform Independent	• Bytecode is platform independent and can be run on any machine, allow security
• Multi-Threading	• utilizes same memory and other resources to execute multiple threads at the same time
• High Performance	• the use of just-in-time compiler
• Distributed	• designed to run on computer networks

1.4 COMMENTS IN JAVA

In Java, comments are used to annotate and explain code, making it easier for developers to read and understand the program. Comments are not executed by the Java compiler, which means they do not affect the program's functionality. There are three types of comments in Java:

- **Single-line comments:** These comments start with // and extend to the end of the line.
- **Multi-line comments:** These comments start with /* and end with */. They can span multiple lines.
- **Documentation comments:** These comments start with /** and are used to generate documentation through tools like Javadoc.

Examples:

- **Single-line comment:**

```
// This is a single-line comment

int number = 10; // This variable stores the number 10
```

- **Multi-line comment:**

```
/*
```

```
 * This is a multi-line comment
 * that can span multiple lines.
 */
int total = 5; /* This variable holds the total value */
```

- **Documentation comment:**

```
/**
 * This method calculates the sum of two integers.
 * @param a An integer value
 * @param b Another integer value
 * @return The sum of a and b
 */
public int sum(int a, int b) {
    return a + b;
}
```

Using comments effectively helps in clarifying the purpose of the code and maintaining it in the long run.

1.5 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 1.2.

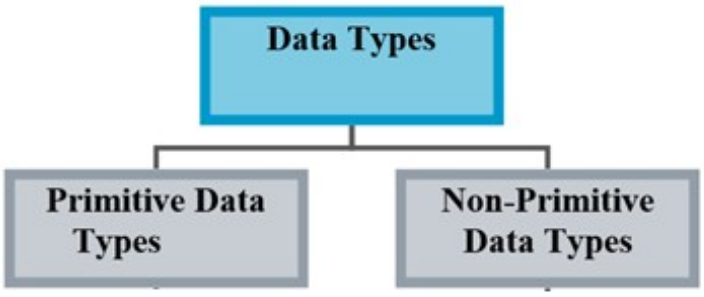


Figure 1.2 Classification of Java Data Types

❖ 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 1.2:

➢ 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 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 ±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 ±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 1.2 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 and are shown in Figure 1.3. Reference variables store the memory address of the object they refer to, rather than the data itself.

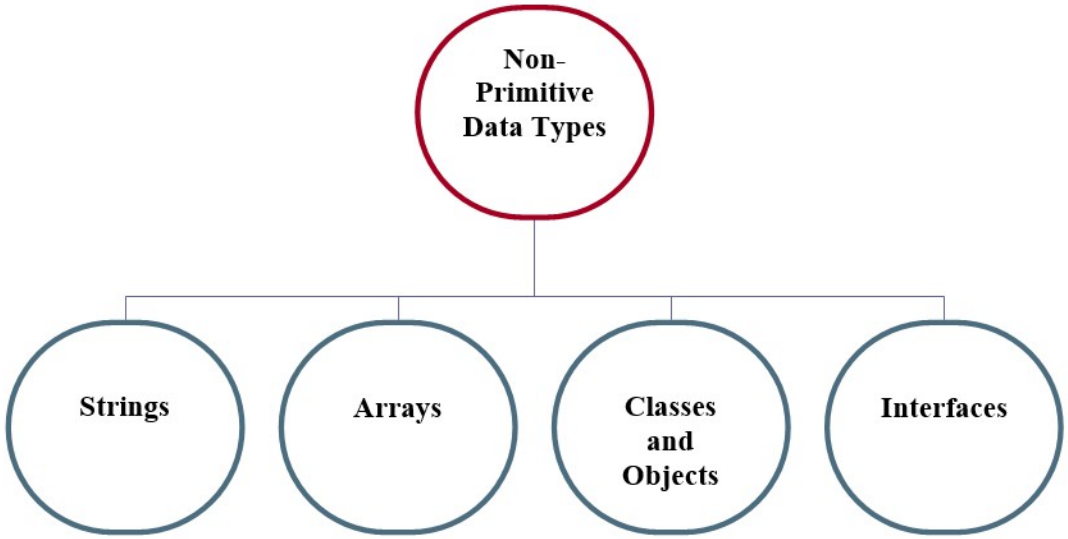


Figure 1.3 Non-Primitive Data Types in Java

- **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:**

Example:

```
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:**

```
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.

1.6 VARIABLES

In Java, a variable is a container that holds data. It is a memory location with a specific type, identified by a name. Variables allow you to store and manipulate values in your program.

- **Declaration of Variables**

Declaration is the process of defining a variable's name and type without assigning it a value.

The syntax for declaring a variable in Java is:

```
dataType variableName;
```

Example:

```
int age; // Declares an integer variable named 'age'.
```

String name; // Declares a String variable named 'name'.

- **Initialization of Variables**

Initialization refers to assigning a value to a declared variable. A variable must be initialized before it is used, particularly for local variables.

Example:

age = 25; // Initializes 'age' with the value 25.

name = "Alice"; // Initializes 'name' with the value "Alice".

You can also declare and initialize a variable in one line:

int age = 25; // Declaration and initialization.

String name = "Alice"; // Declaration and initialization.

- **Types of Variables**

Java variables can be categorized into three primary types, which include local variables, instance variable and static variables and shown in Figure 1.4.

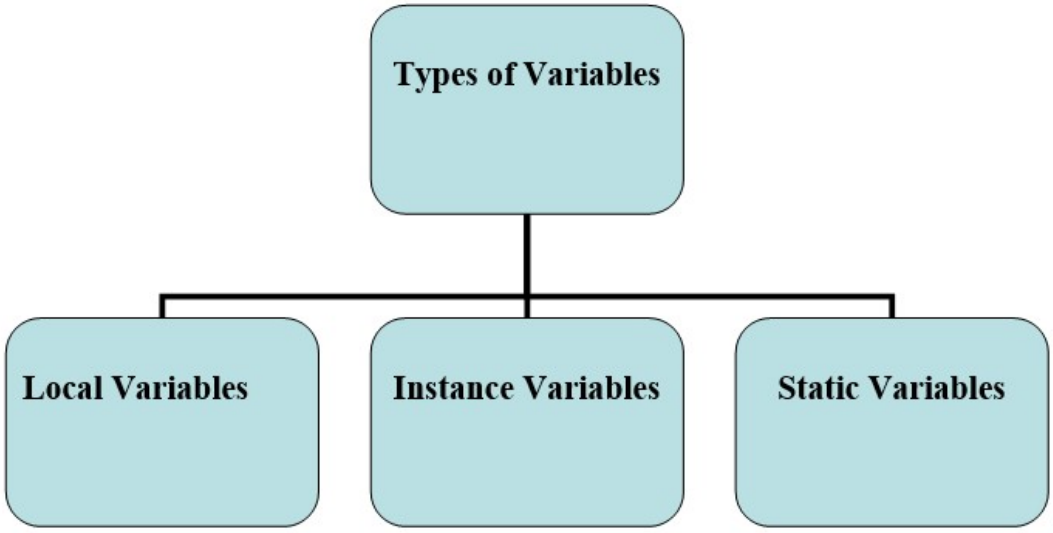


Fig 1.4. Classification of Variables

Local Variables

- Definition: Declared inside a method, constructor, or block. The scope is limited to that method or block.

- Initialization: Must be initialized before use.

Example:

```
public void display() {
   int localVar = 10; // Local variable
   System.out.println("Local Variable: " + localVar);
}
```

Instance Variables

- Definition: Declared within a class but outside any method. Each instance of the class (object) has its own copy of the variable.
- Initialization: Initialized when the object is created or can be set using constructors.

Example:

```
class Person {
   String name; // Instance variable

   Person(String name) { // Constructor to initialize
      this.name = name;
   }
      void display() {
      System.out.println("Name: " + name);
   }
}
public class Main {
   public static void main(String[] args) {
      Person person1 = new Person("Alice");
      person1.display(); // Output: Name: Alice
```

```
        }
    }
```

Static Variables

- Definition: Declared with the static keyword. Static variables are shared among all instances of a class. Only one copy exists for all instances, and it is stored in the static memory.
- Initialization: Can be initialized in the same way as instance variables.

Example:

```
class Counter {
   static int count = 0; // Static variable
   void increment() {
      count++; // Increment the static variable
   }
      void displayCount() {
      System.out.println("Count: " + count);
   }
}
public class Main {
   public static void main(String[] args) {
      Counter counter1 = new Counter();
      Counter counter2 = new Counter();

      counter1.increment();
      counter1.displayCount(); // Output: Count: 1

      counter2.increment();
      counter2.displayCount(); // Output: Count: 2
   }
}
```

1.7 CONSTANTS

In Java, constants are values that do not change during the execution of the program. Constants are typically used when you have values that should remain fixed throughout the program, such as the value of pi, maximum or minimum limits, or configuration values. Constants are typically used to represent fixed values such as mathematical constants or configuration values.

- **Declaration of Constants**

A constant must be declared with the final keyword in Java. This ensures that the value assigned to the constant cannot be changed after initialization. Additionally, constants are typically declared as static when you want them to be shared among all instances of the class.

- **Initialization of Constants**

Constants are initialized at the time of declaration or in the constructor if they are instance constants. However, once a constant is initialized, its value cannot be changed.

- **Types of Constants**

There are two main types of constants in Java:

Instance Constants: These are constants that belong to an instance of the class. Each instance of the class can have different values, but once set, they cannot be changed.They are declared final but without the static keyword.

Example:

```
public class Circle {
    // Instance constant
    public final double radius;
    // Constructor to initialize the constant
    public Circle(double radius) {
        this.radius = radius;
    }
    public double getArea() {
        return MathConstants.PI * radius * radius;
```

```
    }

    public static void main(String[] args) {

        // Creating an instance of Circle

        Circle circle = new Circle(5);

        System.out.println("Area of circle: " + circle.getArea());

    }

}
```

- public final double radius; is an instance constant.
- It is initialized through the constructor and cannot be changed after that.
- The constant PI from the MathConstants class is used to calculate the area of the circle.

Class Constants (Static Constants): These are constants that belong to the class itself rather than any instance. They are declared both final and static. They are often used for values that are common to all objects of that class.

Example:

```
public class MathConstants {

    // Declare a static constant

    public static final double PI = 3.14159;

    public static void main(String[] args) {

        // Accessing the static constant without creating an object

        System.out.println("Value of PI: " + PI);

    }

}
```

- public static final double PI = 3.14159; is a class constant.
- It is declared static, so it can be accessed using the class name (MathConstants.PI) without creating an instance of the class.
- It is declared final, meaning its value cannot be modified after initialization.

- **Constants in Enums**

Enums in Java can also have constant values associated with them. Enums are a special kind of class used to define a set of constant values.

```
public class DayExample {
   public enum Day {
      SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY
   }
   public static void main(String[] args) {
      Day today = Day.MONDAY;
      switch(today) {
         case MONDAY:
            System.out.println("Start of the week!");
            break;
         case FRIDAY:
            System.out.println("End of the week!");
            break;
         default:
            System.out.println("Mid-week.");
      }
   }
}
```

- Day is an enum with constants representing days of the week.
- Each value (SUNDAY, MONDAY, etc.) is a constant, and they cannot be modified.

1.3 Types of constants in Java

Constant Type	Declaration	Initialization	Example Usage
Static Constants	public static final <type> <name> = <value>;	Initialized at declaration or in a static block	public static final double PI = 3.14159;
Instance Constants	public final <type> <name>;	Initialized via constructor	public final double radius;

Enum Constants	enum <name>
	{
	<constant1>, <constant2>, ...
	}
		Declared within the enum type itself	enum Day
			{
			MONDAY, TUESDAY, WEDNESDAY, ...
			}

➢ **Best Practices for Constants**

- **Naming Conventions:** Constants are usually written in uppercase letters with words separated by underscores (_), following the naming convention of UPPER_SNAKE_CASE.
- **Accessibility:** Constants should generally be made public static final to allow for easy access across the program.
- **Value Types:** Constants can be of any primitive type (int, double, boolean, etc.) or reference type (like String).

7. Other types of Constants and Their Use Cases

- **Mathematical Constants:** Constants like Math.PI, Math.E, or any other fixed mathematical value.
- **Configuration Constants:** Constants used for configuration, such as MAX_USER_LIMIT = 1000; or DEFAULT_PORT = 8080;.
- **Application Constants:** Constants used to define fixed application-level values, such as error codes, file paths, or predefined messages.

constants are a useful way to store values that should not change during the execution of a program. Using the final keyword ensures that the value remains constant, and static can be used when the constant is associated with the class itself. Constants help maintain code readability and prevent errors caused by modifying values that should remain fixed.

1.7 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:

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:

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:

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:

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:

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:

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:

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:

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

1.9 Operator Precedence:

In Java, **operator precedence** defines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before those with lower

precedence. If two operators have the same precedence, their **associativity** (whether they are evaluated left-to-right or right-to-left) determines the order.

- **Highest Precedence:**
  - **Array Subscript []** and **Member Reference .**
    - These operators have the highest precedence. For example, in array[i].method(), the subscript i is evaluated first, followed by the method call method().
  - **Postfix ++, --:**
    - Postfix increment and decrement operators (expr++, expr--) have higher precedence than other operators, and they first evaluate the operand before performing the increment or decrement.
    - Example: int x = 5; int y = x++; → x is evaluated first (as 5), and then incremented to 6.
  - **Unary Operators ++, --, +, -, ~, !:**
    - Unary operators such as ++ (increment), -- (decrement), + (positive), - (negative), ~ (bitwise NOT), and ! (logical NOT) come next in precedence.
    - Example: int x = -5; → The unary minus operator - is applied first.
  - **Multiplicative Operators *, /, %:**
    - Multiplication (*), division (/), and modulus (%) operators have higher precedence than the additive operators (+, -).
    - Example: int result = 3 + 2 * 5; → Multiplication (2 * 5) is evaluated first, then addition (3 + 10).
  - **Additive Operators +, -:**
    - Addition and subtraction operators have lower precedence than multiplication, division, and modulus operators.
    - Example: int result = 10 - 5 + 3; → Subtraction (10 - 5 = 5), then addition (5 + 3 = 8).
  - **Shift Operators <<, >>, >>>:**
    - The shift operators are used to shift bits to the left (<<), to the right (>>), and unsigned right (>>>). They come after the additive operators in terms of precedence.
    - Example: int result = 5 << 1; → Left shifts the bits of 5 by 1 position.

- **Relational Operators <, >, <=, >=, instanceof:**
    - Relational operators are used for comparison. instanceof checks whether an object is an instance of a specific class or interface.
    - Example: if (x > y) {...} → Compares if x is greater than y.
- **Equality Operators ==, !=:**
    - The equality operators are used to compare if two values are equal (==) or not equal (!=).
    - Example: if (x == y) {...} → Compares whether x is equal to y.
- **Bitwise Operators &, ^, |:**
    - These operators perform bitwise AND (&), XOR (^), and OR (|) operations on integer values.
    - Example: int result = a & b; → Performs a bitwise AND between a and b.
- **Logical Operators &&, ||:**
    - Logical AND (&&) and logical OR (||) are used for boolean expressions. && has higher precedence than ||.
    - Example: if (a && b) {...} → Evaluates the logical AND of a and b.
- **Ternary Conditional Operator ?::**
    - The ternary conditional operator is used for conditional expressions: condition ? expr1 : expr2. It has higher precedence than logical AND/OR operators.
    - Example: int result = (x > y) ? x : y; → If x > y is true, result gets x, otherwise y.
- **Assignment Operators =, +=, -=, etc.:**
    - Assignment operators, including =, +=, -=, etc., are evaluated last, and they have right-to-left associativity.
    - Example: a = b = 5; → First b = 5 is evaluated, then a = b.
- **Comma ,:**
    - The comma operator is used to separate multiple expressions in a single statement and has the lowest precedence.

- Example: int result = (x = 3, y = 4, x + y); → The expressions are evaluated from left to right, but the final value is x + y.

➢ **Associativity of Operators**

- **Left-to-right associativity**: Most operators (including arithmetic, relational, and bitwise operators) have left-to-right associativity. This means that in expressions involving operators of the same precedence, evaluation occurs from left to right.
  - Example: 3 + 2 - 5 → First 3 + 2 = 5, then 5 - 5 = 0.
- **Right-to-left associativity**: Operators like assignment (=, +=, etc.) and the ternary conditional (?:) have right-to-left associativity. This means that these operators evaluate from right to left.
  - Example: a = b = 5 → First b = 5, then a = b.

1.10 Expressions

In Java, an *expression* is a construct made up of variables, operators, and method calls that evaluates to a single value. Expressions are essential in Java because they form the logic within your code—allowing you to perform calculations, manipulate data, and make decisions.

Here are some common types of expressions in Java:

1. Arithmetic Expressions

Perform basic mathematical operations.

int sum = 5 + 3; *// Addition* int product = 4 * 2; *// Multiplication* double quotient = 10.0 / 4; *// Division (floating-point)*

2. Relational Expressions

Compare values and result in a boolean (true or false).

boolean isEqual = (5 == 5); *// true* boolean isGreater = (10 > 7); *// true* boolean isLessOrEqual = (5 <= 8); *// true*

3. Logical Expressions

Combine boolean expressions with logical operators && (AND), || (OR), and ! (NOT).

boolean result = (5 > 3) && (3 < 10); *// true, both conditions are true* boolean anotherResult = (5 > 7) || (3 < 10); *// true, only one condition needs to be true* boolean notTrue = !(5 == 5); *// false, because 5 == 5 is true*

4. Assignment Expressions

Assign values to variables, often with shorthand operators.

int a = 5; *// Assignment* a += 3; *// Equivalent to: a = a + 3;* int b = 4 * (a + 2); *// Assignment with an expression*

5. Conditional (Ternary) Expressions

Short form of an if-else statement.

int x = 5, y = 10; int max = (x > y) ? x : y; *// max will be 10, since y > x*

6. Method Call Expressions

Calling a method can be part of an expression if the method returns a value.

int length = "Hello".length(); *// length is 5* boolean startsWithH = "Hello".startsWith("H"); *// true*

7. Type Casting Expressions

Convert one data type to another.

double pi = 3.14159;

int intPi = (int) pi; *// Casting double to int, intPi is 3*

above mentioned casting is narrow casting (Explicit Casting), must be done manually by placing the type in parentheses () in front of the value.

Another method is Widening Casting (Implicit Casting), automatically performed by the Java compiler. Converts a smaller data type to a larger data type without any loss of information.No explicit syntax is required.

Example:

```
public class WideningCastingExample {
   public static void main(String[] args) {
      int num = 100;
      double convertedValue = num; // Automatic conversion from int to double
      System.out.println("Converted value: " + convertedValue); // Output: 100.0
   }
}
```

Rules for Widening Casting: From smaller to larger types:

- byte → short → int → long → float → double

Example of a Complex Expression

Combining multiple expressions:

int a = 5;

int b = 10;

boolean isAGreater = (a + b > 10) && (a * b < 100); *// true*

In this example, (a + b > 10) and (a * b < 100) are relational expressions that return true or false. The && operator is a logical expression combining both, resulting in a final boolean value.

1.11 SUMMARY

Java is a widely used, object-oriented programming language designed for portability, security, and high performance. Java programs 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. Constants in Java allow users to

restrict a value throughout entire program. Additionally, the operator precedence, and how it work on various expressions also provided.

1.12 TECHNICAL TERMS

- Data Type
- Operator
- static
- Boolean
- Type casting
- arithmetic
- operator precedence

1.13 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain the structure and significance of data types in Java programming.
2. Describe the concept of operator precedence with example java program.
3. Analyze the key buzzwords 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 various types of constants in Java.

Short questions:

1. List and briefly describe expressions in Java.
2. What is the purpose of enum datatype in Java?
3. What is the role of data types in Java?

1.14 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

AUTHOR: Dr. KAMPA LAVANYA

LESSON- 02

OOPS CONCEPTS

AIMS AND OBJECTIVES

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

1. Understand the basic concepts of Object-Oriented Programming (OOP).
2. Recognize the applications of OOP
3. Explain the fundamental features of OOP, including encapsulation, inheritance, polymorphism, and abstraction.
4. Apply the principles of OOP to design and implement modular and maintainable software solutions.

STRUCTURE

2.1 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. Also explained each concept detailly with examples. In addition, constructors and parameter passing techniques also discussed.

2.2 OBJECT-ORIENTED PROGRAMMING (OOP)

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.

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 2.1.

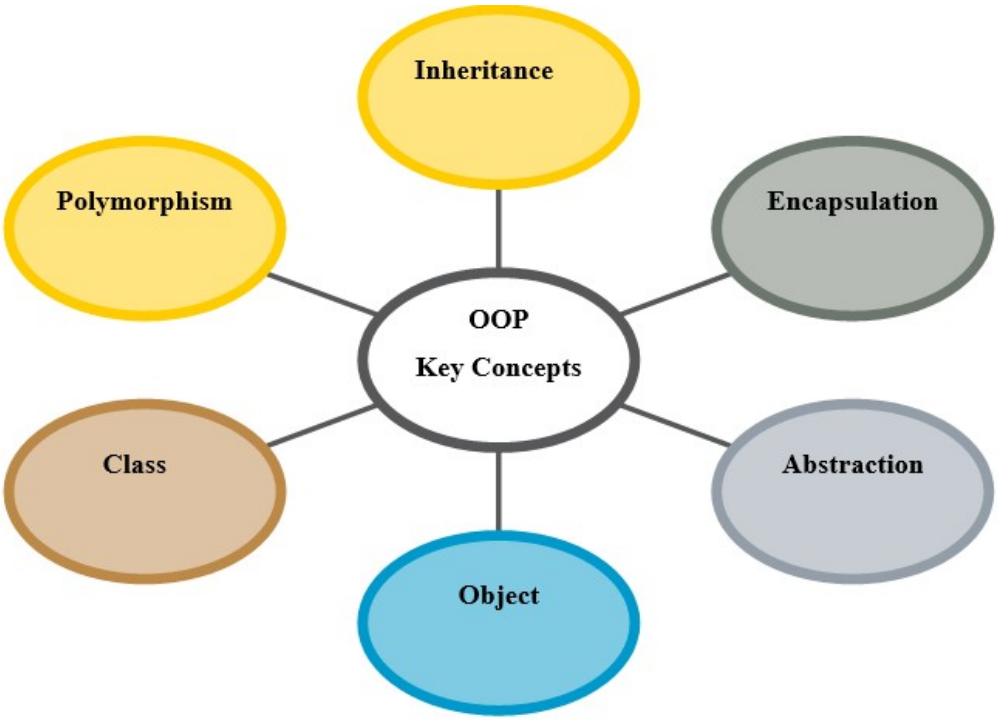


Fig 2.1. Key Concepts of OOP

❖ 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 2.2.

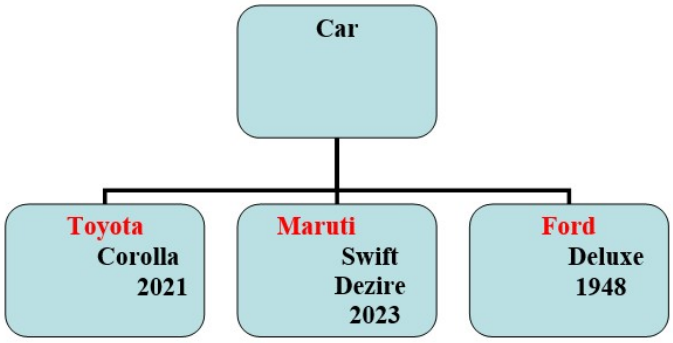


Fig 2.2. Object concept in OOP

❖ 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 2.3.

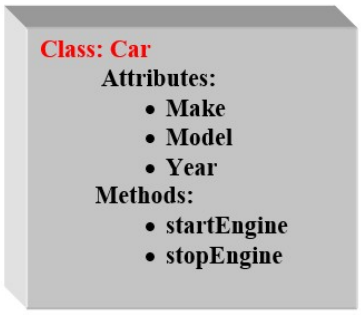


Fig 2.3. Class concept in OOP

❖ **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 2.4.

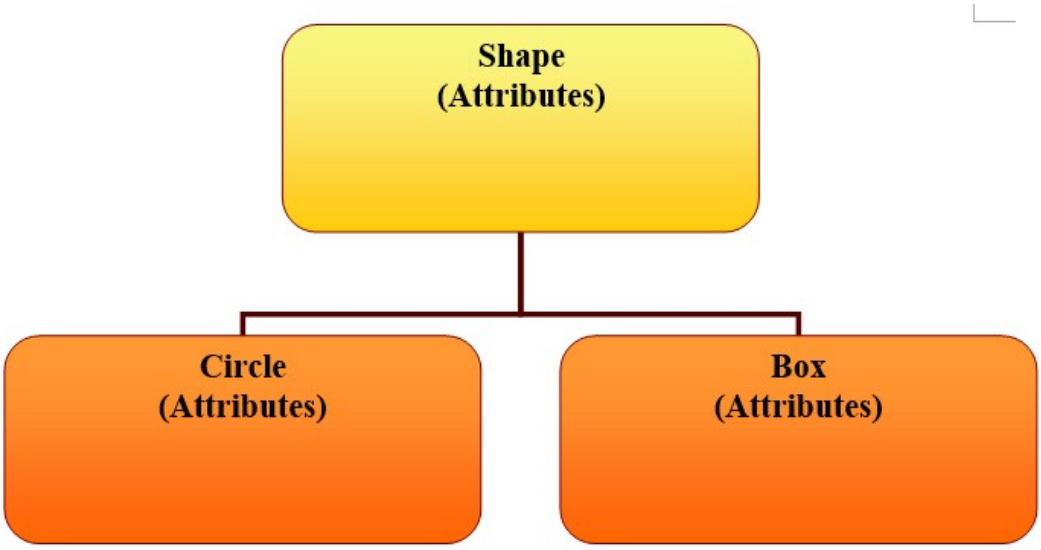


Fig 2.4. Inheritance concept in OOP

❖ **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 2.5.

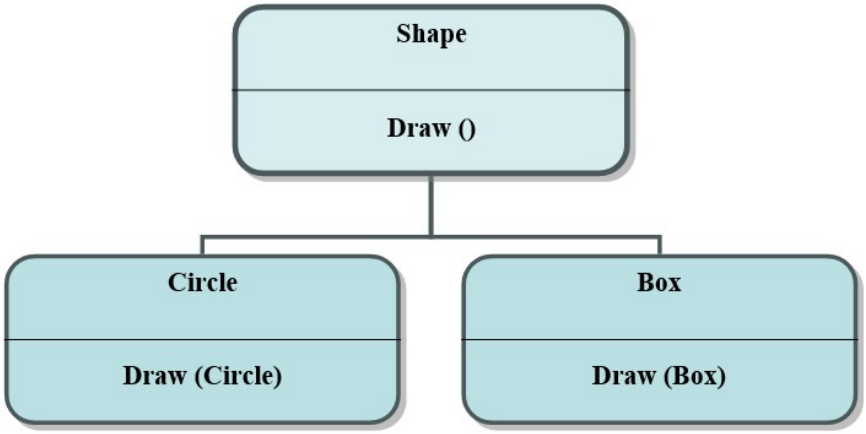


Fig 2.5. Polymorphism Concept in OOP

❖ **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.

Here are the key features of Object-Oriented Programming (OOP) listed in a concise, point-wise format and shown in Table 2.1:

Table 2.1 Features of OOP

Feature	**Description**
Encapsulation	Bundles data (attributes) and methods (functions) into a single unit (class) and restricts access to some components to protect data integrity.
Abstraction	Hides complex implementation details and exposes only the necessary parts, allowing focus on what an object does rather than how it does it.
Inheritance	Allows a new class to inherit properties and behaviors from an existing class, promoting code reusability and reducing redundancy.
Polymorphism	Enables objects of different classes to respond to the same function call in different ways, enhancing flexibility and scalability.

Modularity	Encourages the division of a program into smaller, self-contained modules (classes), improving code organization, readability, and maintainability.
Reusability	Facilitates the reuse of existing code in new applications, saving time and reducing errors.
Dynamic Binding	Determines the method to be invoked at runtime, providing flexibility and supporting polymorphism.
Message Passing	Objects communicate by sending messages (function calls) to each other, enabling complex behaviors through object interactions.

2.3 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.

2.4 CLASS

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.

❖ **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
}
```

❖ **Instance Variables**

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;
}
```

❖ **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 void getInfo(String model, int year) {
      model = model;
      year = year;
   }
   // Method
```

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

❖ **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.

Example:

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

OBJECT

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.

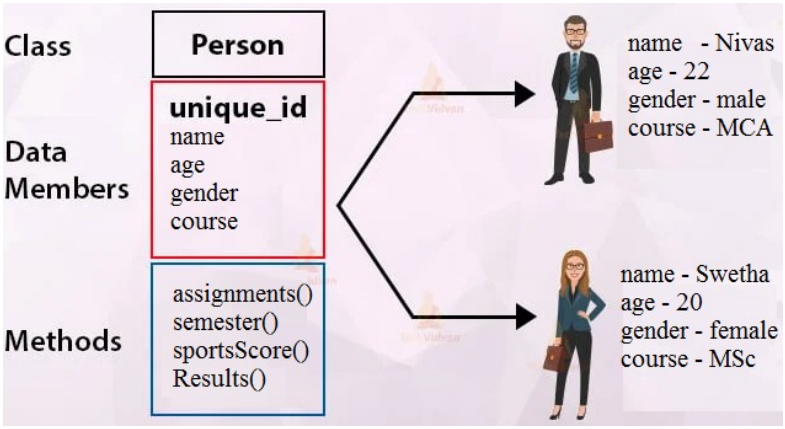


Figure 2.6 class and objects

❖ **Declaring and Instantiating Objects in Java**

creating a new instance of P and initializing its name to Nivas, age to 22 , gender to male , and course to MCA.

Example:

```
Class Person
{
        private string name;
        private int age;
        private string gender;
        private string course;
          Public  Get_Person( String name,int age,Stirng gender,String
course,double sscore)
                {
                   this.name =name;
                   this.age = age;
                   this.gender = gender;
                   this.course = course;
                   this.sscore = sscore;
            }
                public void displayPerson() {
                    System.out.println("Name: " + name);
                    System.out.println("Age: " + age);
                    System.out.println("Gender: " + gender);
                    System.out.println("Course: " + course);
            }
             Public void sportsScore()
                    {
                        System.out.println("Sport Score: " + sscore);
                    }
   }
   class Main{
    public static void main(String [] args){
            Person P= new Person()
             P.Get_Person( “Nivas”, “22 “, “male” , “MCA”);
```

```
            P. displayPerson();
           P.sportsScore();
            }
      }
```

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

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();
   }
}
```

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();
    }
}
```

2.5 ACCESS CONTROL

Access Control in Java imposes limitations on the extent of the class, for instance variables, methods, and constructor's scope.

In Java, there are 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 the 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.

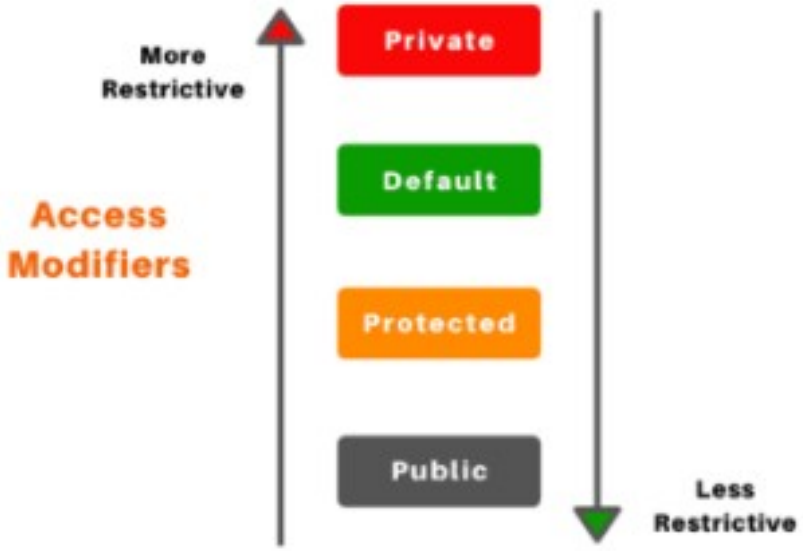


Figure 2.7 types of access modifiers

2.6.1 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 private package.

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 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\.jdks\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.

2.6.2 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 ob1 = 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
```

2.6.3 . 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.

2.6.4 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 2.8 summary of access modifiers

2.6 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.

2.6.1 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.

2.6.2 Types of Constructors

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

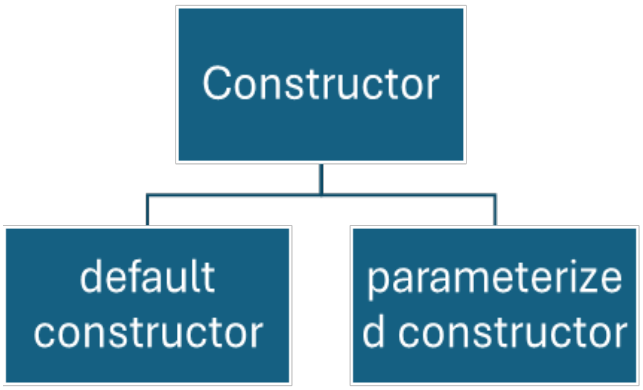


Figure 2.9 types of constructors

2.7.2.1 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
    }
}
```

2.7.2.2 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
    }
}
```

2.7.3 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
    }
}
```

2.7.4 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 {
ring 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

   }

}
```

2.8 PARAMETER PASSING

Parameter Passing in java can be done with either

1. Call by Value
2. Call by Reference

- Call by Reference (Conceptual, Not Used in Java)
- In Call by Reference, a method receives a reference to the actual variable.
- Changes made inside the method reflect in the original variable.
- Java does not use Call by Reference, but it can sometimes appear similar when dealing with objects.
- Reference Types in Java (Objects)
- When passing objects (like arrays, StringBuilder, custom objects), the reference (address) is passed by value.

- This allows methods to modify the state of the object but not the reference itself. The reference sb is passed by value, so reassigning it in the method does not affect the original reference.
  Example: Modifying Object State
  Java
  Copy code
  class CallByValueExample {
  public static void modifyObject(StringBuilder sb) {
  sb.append(" World!"); // Modifies the object's state
  }
  public static void main(String[] args) {
  StringBuilder sb = new StringBuilder("Hello");
  modifyObject(sb);
  System.out.println(sb); // Output: Hello World!
  }
  }
- The reference to sb is passed by value. The method accesses the same object and modifies its state.
- However, reassigning the reference inside the method does not affect the original object.

```
class CallByValueExample {
   public static void reassignReference(StringBuilder sb) {
      sb = new StringBuilder("New Object"); // Changes only the local reference
   }

   public static void main(String[] args) {
      StringBuilder sb = new StringBuilder("Hello");
      reassignReference(sb);
      System.out.println(sb); // Output: Hello
   }

}
```

Table 2.2 Summary of Parameter Passing

Type	Technique	Behavior
Primitive Types	Call by Value	Copies the value, changes inside the method do not affect the original value.
Reference Types	Call by Value (with references)	Copies the reference, changes to the object's state affect the original, but reassigning does not.

2.9 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.

2.10 TECHNICAL TERMS

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

2.11 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.
6. Describe about Parameter Passing

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?

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

AUTHOR: Dr. KAMPA LAVANYA

LESSON- 03

CONDITIONAL STATEMENTS

AIMS AND 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

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 FEATURES 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.

Features:

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:

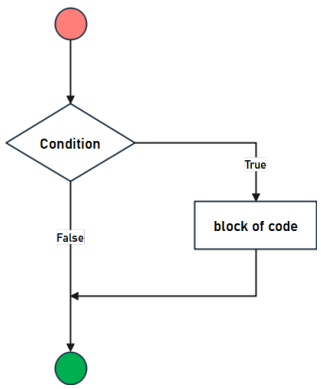


Fig 3.1. Flow Chart of if Statement

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

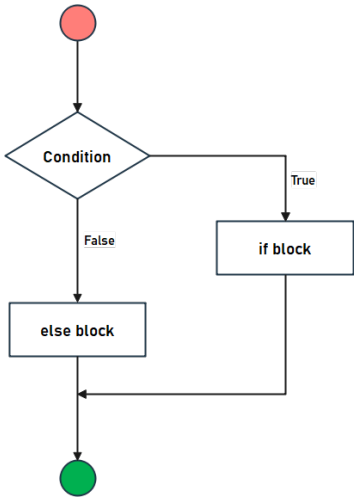


Fig 3.2. Flow Chart of if-else Statement

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:

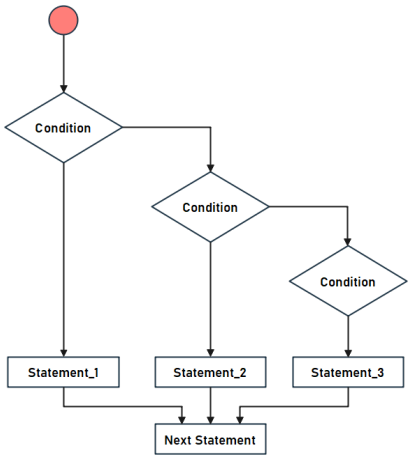


Fig 3.3. Flow Chart of else-if ladder Statement

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:

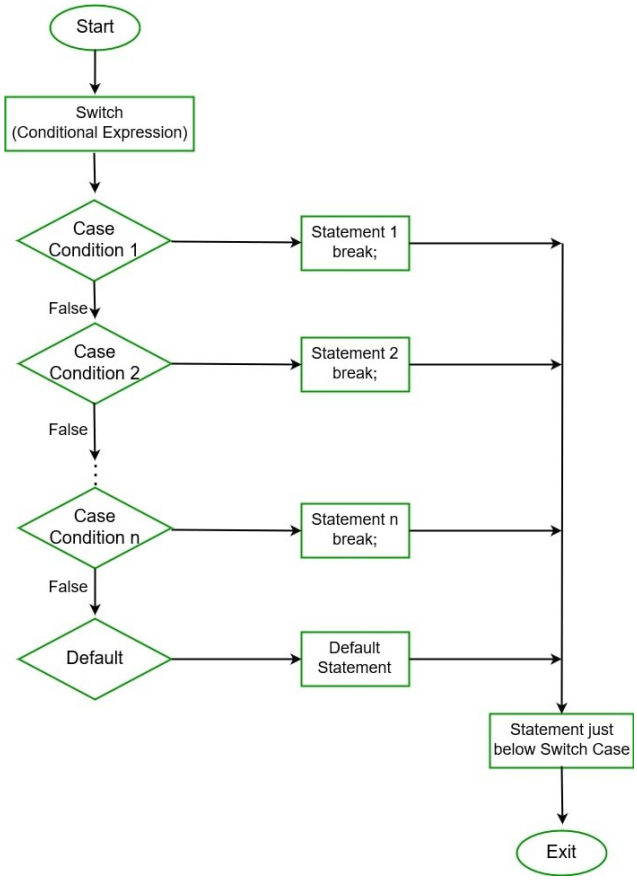


Fig 3.4 Flow Chart of else-if ladder Statement

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

AUTHOR: Dr. KAMPA LAVANYA

LESSON- 04

LOOP STATEMENTS

AIMS AND 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

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:

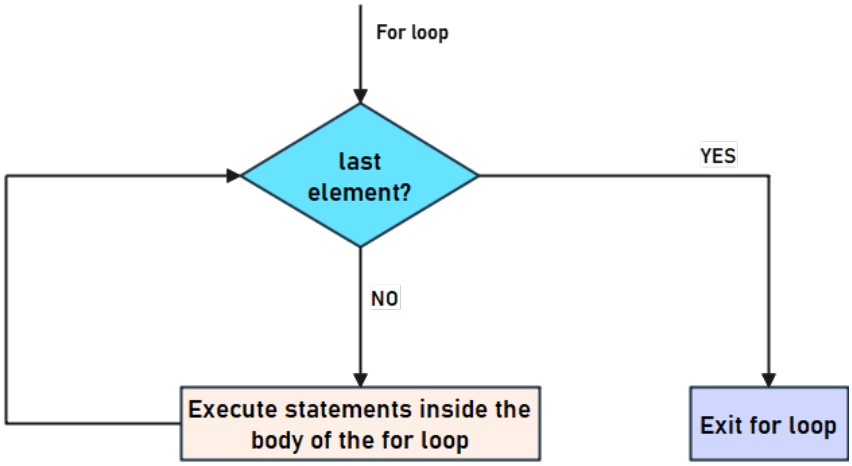


Fig 4.1. Flowchart of For-loop Statement

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:

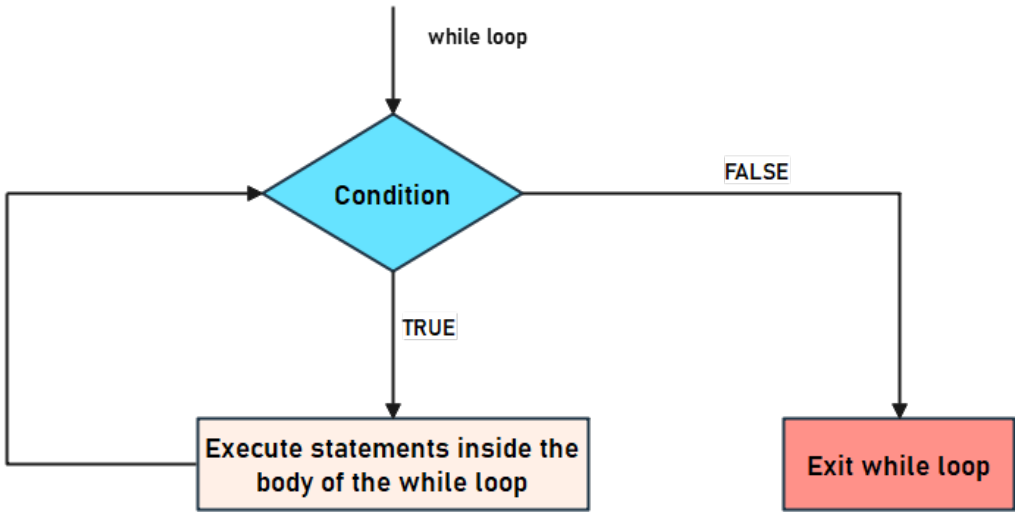


Fig 4.2 Flowchart of While-loop Statement

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

Thes loops are 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.

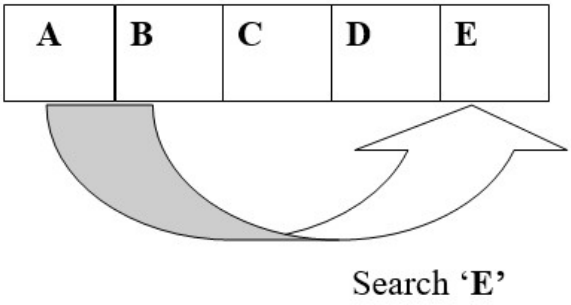


Fig 4.3. Linear Search Example

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));
    }
}
```

- **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
• In linear search, input data doesn't need to be sorted .	• Whereas, in binary search, input data has to be sorted according to the order.
• It is also referred as sequential search.	• It is also referred to as half-interval search.
• The time complexity of the linear search is O(n)	• The time complexity of the binary search is 0 (logn)
• Multi-dimensional array is used for linear search.	• A single dimensional array is used for linear search.
• It operates equality comparisons	• Binary search operates ordering comparisons

• Linear search is less complex and involves a slow process	• 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.

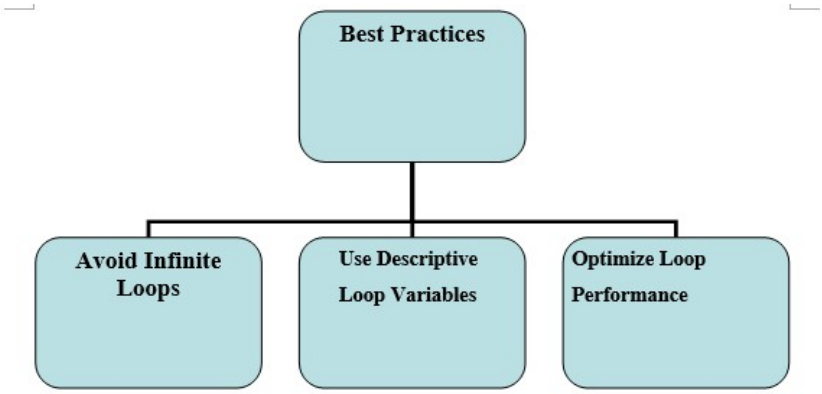


Fig 4.4 Best Practices For Loop Statements

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

AUTHOR: Dr. KAMPA LAVANYA

LESSON- 5

ADVANCE JAVA CONCEPTS

OBJECTIVES:

After going through this lesson, you will be able to

- Learn about various advanced java concepts
- Understand about array concepts
- Explore about formatting output
- 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.
- Understand the concept of recursion.

STRUCTURE OF THE LESSION:

5.1 INTRODUCTION

This chapter introduces key Java programming concepts, beginning with Formatting Output, which demonstrates methods like printf and String.format to produce well-structured output. The topic of Arrays explores storing collections of similar data, detailing Types of Arrays such as single-dimensional and multi-dimensional arrays, and their Creation through declaration, initialization, and access. The Operations on Arrays section covers essential tasks like traversal, sorting, and searching. Moving to Strings, the chapter explains their immutable nature, supported by String Class Methods for manipulation, and techniques for String Comparison using equals(), ==, and compareTo(). Recursion introduces solving problems by having methods call themselves, while the this Keyword clarifies referencing the current object. Garbage Collection highlights Java's automatic memory management, ensuring unused objects are removed, and Autoboxing and Unboxing describes the seamless conversion between primitives and their wrapper classes. Together, these topics provide a comprehensive understanding of Java's core functionalities.

5.2 FORMATTED OUTPUT

Formatted output in Java allows you to control how data is displayed, ensuring readability and precision. Java provides several mechanisms for formatting output, primarily through the printf and String.format methods, which follow a similar syntax.

❖ printf Method

The printf method, part of java.io.PrintStream, is used to format output directly to the console.

Syntax:

System.out.printf(formatString, arguments);

- **formatString**: Specifies the format.
- **arguments**: The values to be formatted.

Example:

```
public class FormattedOutputExample {

   public static void main(String[] args) {

      int number = 123;

      double pi = 3.14159;

      String name = "Alice";
```

```
        System.out.printf("Integer: %d\n", number); // Output: Integer: 123
        System.out.printf("Float: %.2f\n", pi); // Output: Float: 3.14
        System.out.printf("Name: %s\n", name); // Output: Name: Alice
    }
}
```

❖ **Placeholders in Format Strings**

Placeholders are used to define how values will be formatted.

Table 5.1 Placeholders in Format Strings

Specifier	**Description**
%d	Decimal integer
%f	Floating-point number
%.nf	Floating-point with n decimal places
%s	String
%c	Character
%x	Hexadecimal integer
%o	Octal integer
%e	Scientific notation (e.g., 1.23e+03)

Example with Multiple Placeholders:

```
System.out.printf("Name: %s, Age: %d, GPA: %.2f\n", "Bob", 20, 3.75);
// Output: Name: Bob, Age: 20, GPA: 3.75
```

❖ String.format Method

This method formats a string without printing it directly. It is useful for creating formatted strings for further use.

Syntax:

String formattedString = String.format(formatString, arguments);

Example:

```
public class StringFormatExample {
   public static void main(String[] args) {
      int age = 25;
      String message = String.format("I am %d years old.", age);
      System.out.println(message); // Output: I am 25 years old.
   }
}
```

❖ Flags for Formatting

Flags modify the output format to include alignment, padding, or special symbols.

Table 5.2 Flags for Formatting

Flag	Description	Example
-	Left-align the output	System.out.printf("%-10s", "Hello");
+	Include a sign for numbers	System.out.printf("%+d", 123); // +123
0	Pad with zeros	System.out.printf("%05d", 42); // 00042
,	Include grouping separators	System.out.printf("%,d", 1000000); // 1,000,000
(	Enclose negative numbers in parentheses	System.out.printf("%(d", -42); // (42)

❖ **Formatting Dates and Times**

The printf and String.format methods also support date and time formatting using the %t or %T specifiers.

Example:

```
import java.util.Date;

public class DateFormattingExample {

   public static void main(String[] args) {

      Date today = new Date();

      System.out.printf("Current date: %tF\n", today); // Output: YYYY-MM-DD

      System.out.printf("Current time: %tT\n", today); // Output: HH:MM:SS

   }

}
```

5.3. ARRAYS

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.

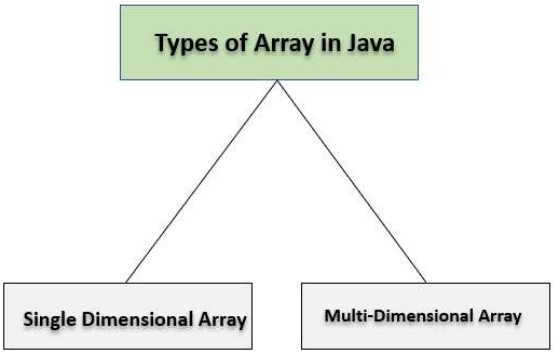
5.4. TYPES OF ARRAYS

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



5.1 types of arrays in Java

❖ One-dimensional array

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:

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

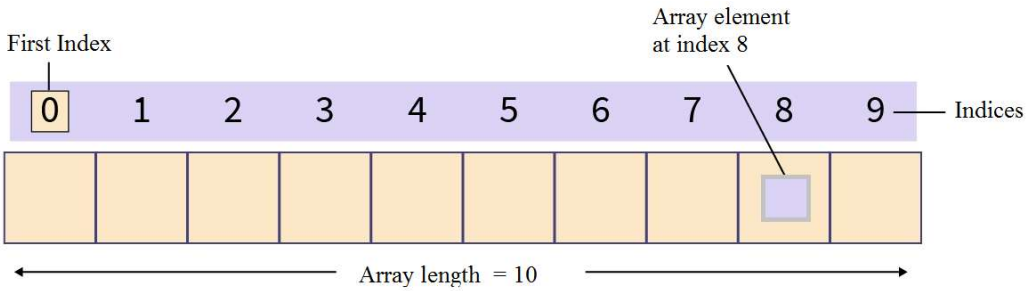


Figure 5.2 Single - dimensional array representation

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

❖ 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.

➢ 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
```

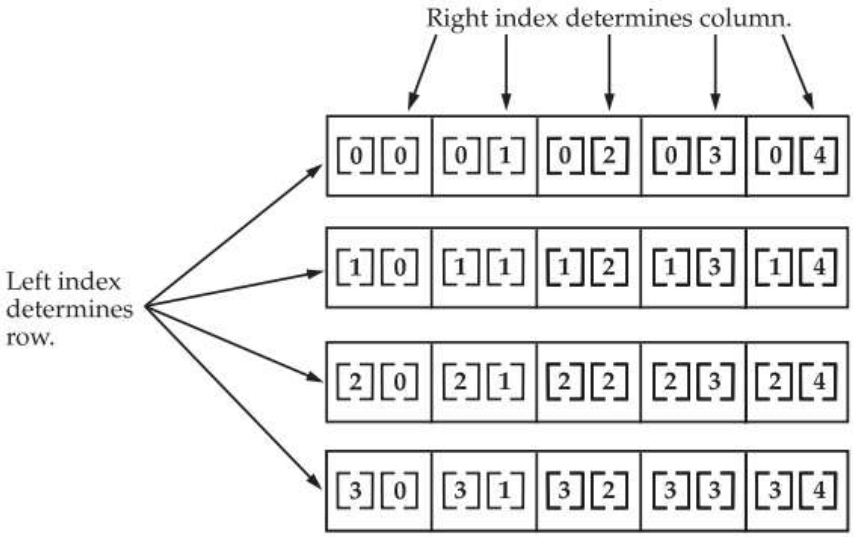


Figure 5.3 structure of multi-dimensional array

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.

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
	}
	}
	}
```

5.5 OPERATION PERFORMED ON ARRAYS

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

- **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
```

❖ 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);
}
```

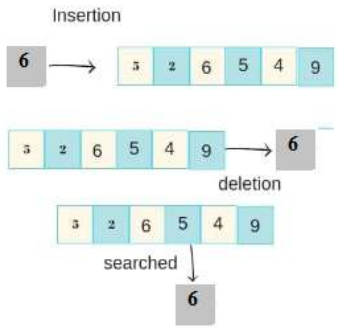
❖ Finding the Length of an Array

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

❖ **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);
}
```

❖ **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

❖ **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

❖ **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();
}
```

❖ **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");
}
```

5.6 STRINGS

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.

❖ Creating Strings

There are several ways to create strings in Java:

- **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!";
```

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

Example:

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

- **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 RECURSION

Recursion is a programming technique where a method calls itself directly or indirectly to solve a problem. Each recursive call works on a smaller subproblem until a base condition is met, at which point the recursion stops. Recursion is often used for problems that can be broken down into similar subproblems, such as mathematical computations, searching, sorting, and traversing data structures.

Key Components of Recursion

- Base Case: The condition that stops the recursion. Without it, the recursion will go into an infinite loop, causing a stack overflow error.
- Recursive Case: The part of the method that breaks the problem into smaller subproblems and calls itself.

Example: Factorial Calculation

The factorial of a number n

```
public class RecursionExample {

   public static int factorial(int n) {

      // Base case: factorial of 0 is 1

      if (n == 0) {

         return 1;

      }

      // Recursive case

      return n * factorial(n - 1);
```

```
    }

    public static void main(String[] args) {

        int number = 5;

        System.out.println("Factorial of " + number + " is: " + factorial(number));

    }

}
```

Output:

Factorial of 5 is: 120

5.11 THIS KEYWORD

The this keyword in Java is a reference variable that refers to the current object of a class. It is used to resolve ambiguity between class attributes and parameters, invoke other constructors, or pass the current object to a method or constructor. The this keyword is one of the most commonly used constructs in object-oriented programming.

1. Referencing Instance Variables

When local variables in a method or constructor have the same name as instance variables, the this keyword distinguishes between them.

Example:

```
class Example {

    int value;

    // Constructor

    Example(int value) {

        this.value = value; // Refers to the instance variable

    }

    void display() {

        System.out.println("Value: " + this.value);

    }

}
```

```
public class Main {
    public static void main(String[] args) {
        Example obj = new Example(10);
        obj.display(); // Output: Value: 10
    }
}
```

2. Calling Another Constructor

The this() syntax can be used to call another constructor in the same class. This is known as **constructor chaining**.

Example:

```
class Person {
    String name;
    int age;
    // Constructor 1
    Person(String name) {
        this(name, 0); // Calls Constructor 2
    }
    // Constructor 2
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
public class Main {
```

```
    public static void main(String[] args) {
        Person p = new Person("Alice");
        p.display(); // Output: Name: Alice, Age: 0
    }
}
```

3. Passing the Current Object

The this keyword can be used to pass the current object to another method or constructor.

```
class A {
    void show(B b) {
        b.display(this);
    }
}
class B {
    void display(A a) {
        System.out.println("Method called using current object.");
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.show(b); // Output: Method called using current object.
    }
}
```

4. Returning the Current Object

The this keyword can be used to return the current object from a method. This is useful for method chaining.

Example:

```
class Example {
    int value;
    Example setValue(int value) {
        this.value = value;
        return this; // Returns the current object
    }
    void display() {
        System.out.println("Value: " + value);
    }
}
public class Main {
    public static void main(String[] args) {
        Example obj = new Example();
        obj.setValue(20).display(); // Method chaining
    }
}
```

5.12 GARBAGE COLLECTION

Garbage Collection (GC) in Java is an automatic process that helps manage memory by removing objects that are no longer in use or referenced. This helps in freeing up memory and prevents memory leaks, ensuring efficient use of system resources. Java's garbage collector (GC) is part of the Java Virtual Machine (JVM), which automatically handles memory management.

While garbage collection is mostly automatic, you can suggest that the JVM perform garbage collection using System.gc(), but it's not guaranteed that the garbage collector will run immediately or at all.

System.gc(); // Suggests that the JVM performs garbage collection

However, relying on System.gc() is discouraged as the JVM decides when to run garbage collection based on its own internal algorithms.

Finalization in Java

- **finalize() method**: Prior to garbage collection, an object may override the finalize() method to perform cleanup tasks, such as closing resources (files, network connections, etc.).
- **Note**: The finalize() method is deprecated in modern Java versions due to its unpredictability and performance issues. It's better to use **try-with-resources** or explicitly close resources.

Example:

```
class MyClass {
   @Override
   protected void finalize() throws Throwable {
      System.out.println("Cleanup before object is garbage collected.");
   }
}
public class GarbageCollectionExample {
   public static void main(String[] args) {
      MyClass obj = new MyClass();
      obj = null; // Object is now eligible for garbage collection
      System.gc(); // Suggests garbage collection
   }
}
```

Advantages of Garbage Collection

1. **Automatic Memory Management**: Developers do not need to manually manage memory allocation and deallocation, reducing the chances of memory leaks.
2. **Improved Performance**: By reclaiming memory that is no longer in use, the GC helps improve overall system performance.

3. **Prevents Memory Leaks**: Unused objects are automatically cleared, preventing memory from being consumed by orphaned objects.

Disadvantages of Garbage Collection

1. **Performance Overhead**: Garbage collection consumes system resources, which may cause occasional pauses (known as **GC pauses**) in an application, especially with large heap sizes.
2. **Unpredictability**: The exact time when garbage collection occurs is not predictable, and thus developers may not have control over when resources are freed.
3. **Dependence on JVM**: The garbage collection process is JVM-specific, meaning the exact implementation can vary depending on the JVM being used.

5.13AUTOBOXING AND UNBOXING

Autoboxing and Unboxing are features in Java that allow for the automatic conversion between primitive types and their corresponding wrapper classes (e.g., int to Integer, double to Double, etc.). These conversions make it easier to work with primitive types and objects, especially in situations like collections (e.g., ArrayList) that can only hold objects.

❖ Autoboxing

Autoboxing refers to the automatic conversion of a primitive type into its corresponding wrapper class (i.e., object). This process happens automatically when a primitive type is assigned to a variable of a wrapper class type, or when a primitive value is passed as an argument to a method that expects an object.

```
public class AutoboxingExample {

   public static void main(String[] args) {

      int primitiveInt = 5;

      // Autoboxing: primitive int is automatically converted to Integer object

      Integer wrapperInt = primitiveInt;

      System.out.println("Wrapper Integer: " + wrapperInt);  // Output: Wrapper Integer: 5

   }

}
```

In this example, the primitive int is automatically converted into an Integer object. The compiler handles the conversion automatically, making it easier to work with primitive types as objects.

❖ **Unboxing**

Unboxing refers to the automatic conversion of a wrapper class object into its corresponding primitive type. This happens when a wrapper object is assigned to a variable of the corresponding primitive type, or when a wrapper object is passed to a method expecting a primitive type.

```
public class UnboxingExample {

   public static void main(String[] args) {

      Integer wrapperInt = 10; // Autoboxing: int to Integer

      // Unboxing: Integer object is automatically converted to primitive int

      int primitiveInt = wrapperInt;

      System.out.println("Primitive int: " + primitiveInt);  // Output: Primitive int: 10

   }

}
```

In this example, the Integer object is automatically converted to a primitive int. This conversion is handled by Java automatically, simplifying the process.

5.14 SUMMARY

In conclusion, the topics explored in Java, such as Formatting Output, Arrays, Strings, and Recursion, highlight essential features that enhance programming efficiency and effectiveness. Arrays provide structured data storage, with various types and creation methods, while operations on arrays facilitate efficient data manipulation. Strings in Java are immutable objects, and understanding their class methods, comparison, and immutability is crucial for working with text. Recursion offers a powerful technique for solving complex problems by breaking them into simpler subproblems, and the this keyword helps manage instance variables and object references. Garbage Collection automates memory management, ensuring efficient resource usage, and Autoboxing and Unboxing streamline the interaction between primitive types and their wrapper classes. These concepts collectively make Java a versatile and robust language, offering both simplicity and flexibility for developers.

5.15 TECHNICAL TERMS

Java Array, single – dimension, multi – dimension, String, Immutability,Recursion, this keyword.

5.16 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.
5. Describe about this keyword.

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. Write about Garbage collection.

5.17 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

AUTHOR: Dr. U. Surya Kameswari

LESSON- 06

INHERITANCE

AIMS AND 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

6.1 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.

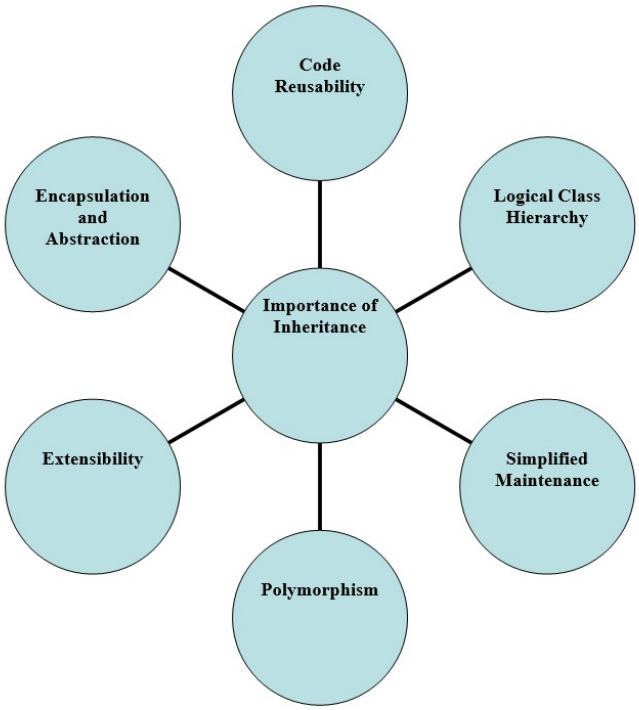


Fig 6.1 Importance of Inheritance

6.2 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");
   }
}
```

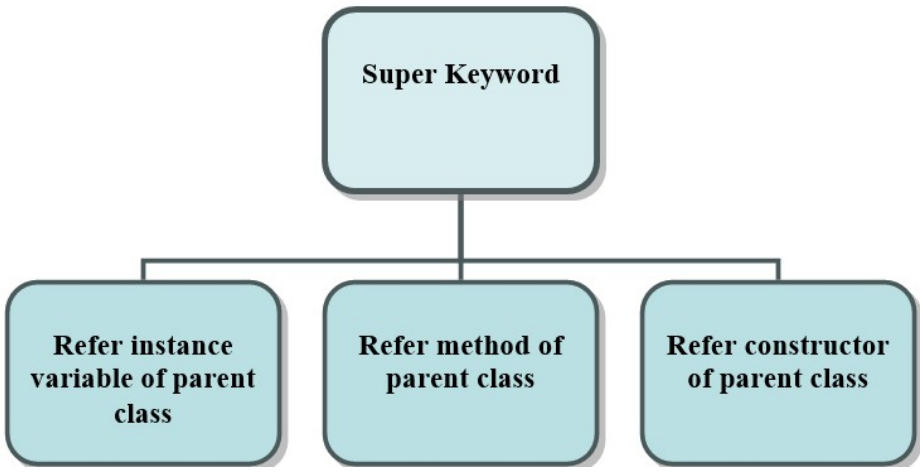


Fig 6.2 Super Keyword in Java

❖ **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.

6.3 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.

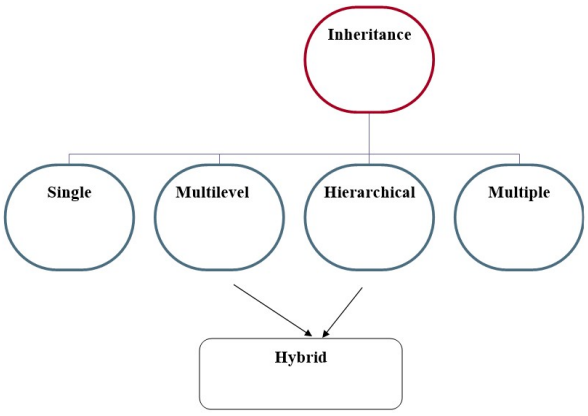


Fig 6.3 Types of Inheritance

❖ 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 6.1 Diffrences 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

6.4 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.

6.5 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:

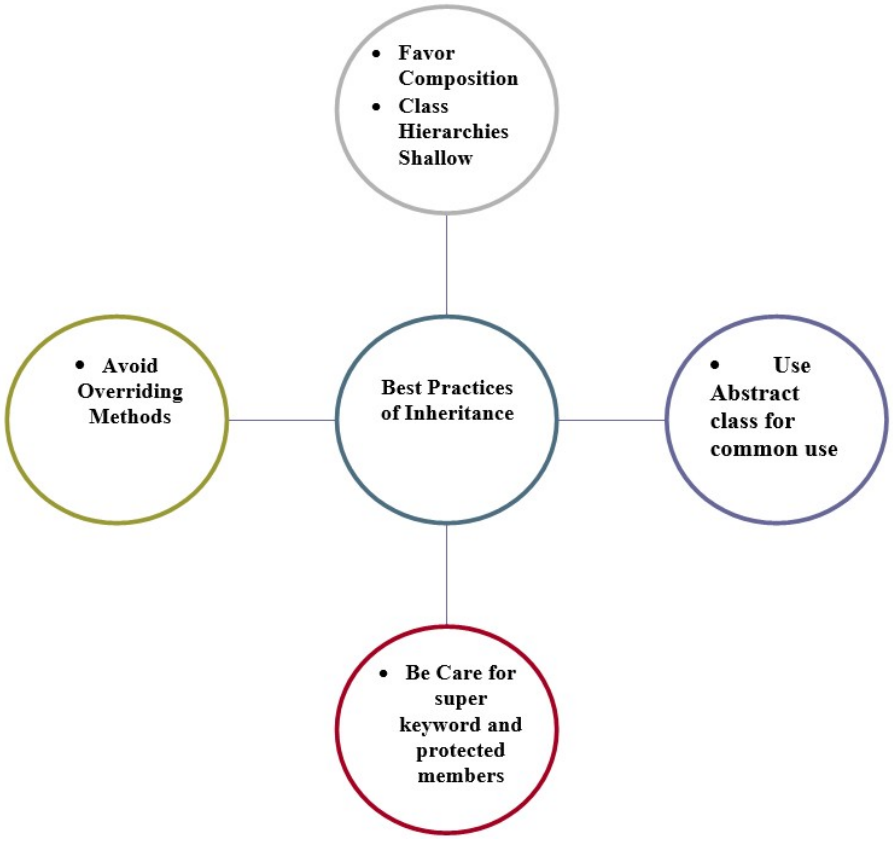


Fig 6.4 Best Practices of Inheritance

❖ **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.

6.6 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.

6.8 TECHNICAL TERMS

1. Super
2. extends
3. single level
4. Hybrid
5. Multiple
6. Method Overriding

6.9 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?

6.10 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

AUTHOR: Dr. U. Surya Kameswari

LESSON- 07

POLYMORPHISM

AIMS AND 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

7.1 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.

7.2 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.

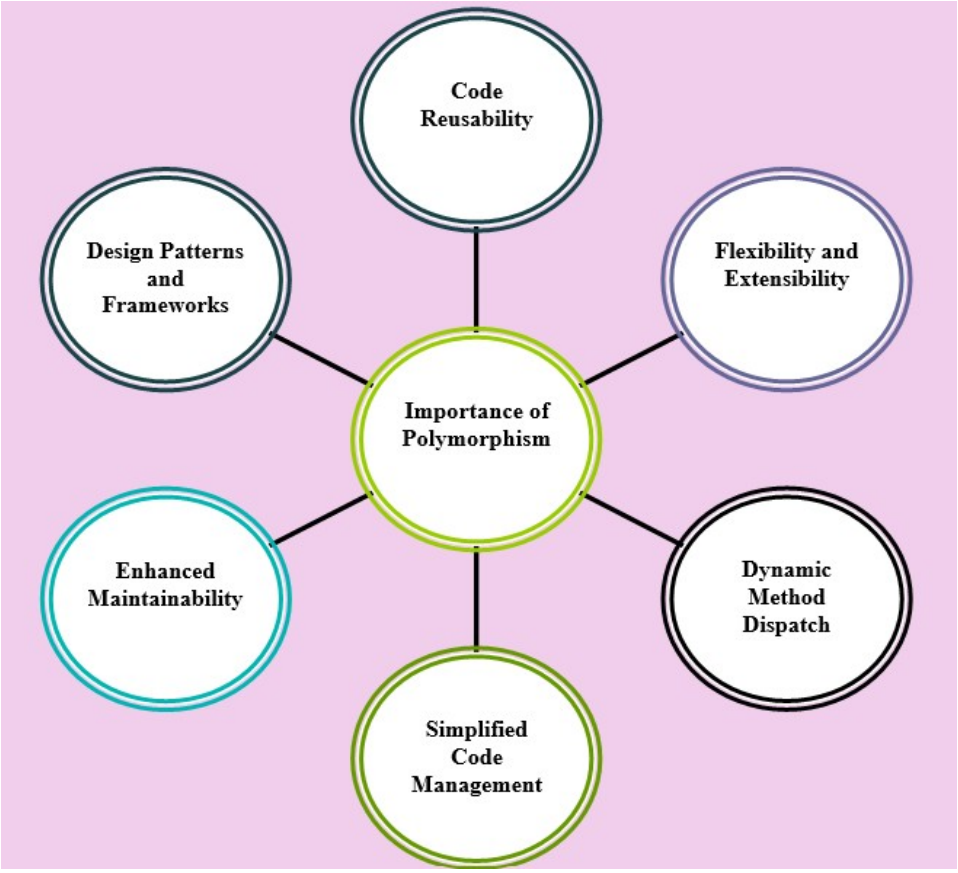


Fig 7.1 Importance of Polymorphism with various factors

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");
   }
```

7.3 COMPILE-TIME POLYMORPHISM

Polymorphism can be classified into two types and described in Figure 8.2.

1. Compile-Time Polymorphism
2. Run-Time Polymorphism

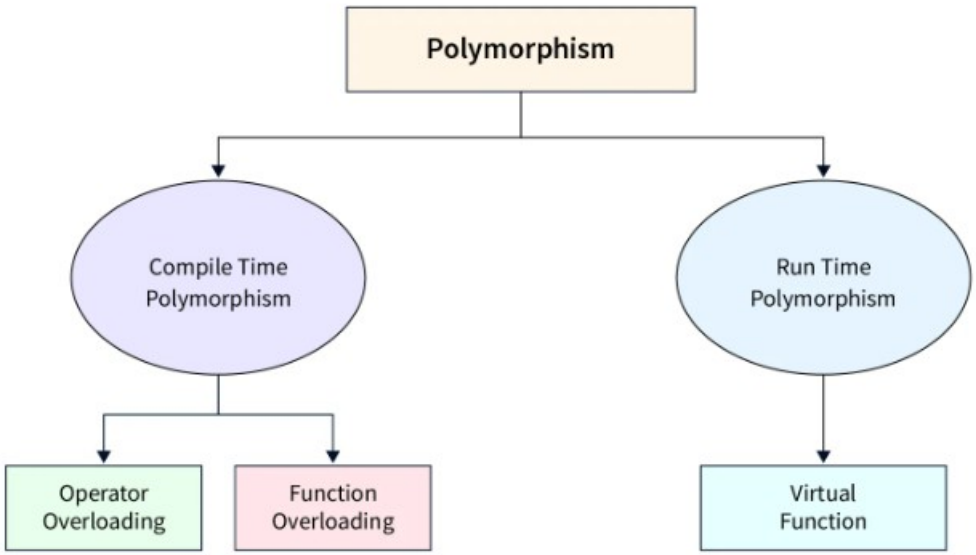


Fig 7.2 Polymorphism classification in Java

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.

7.4 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.

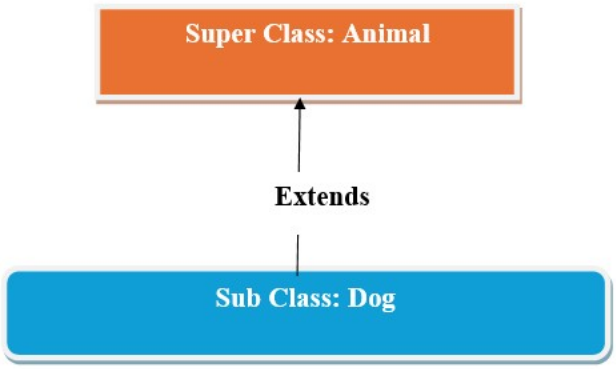


Fig 7.3 Dynamic Dispatch Method for Animal -Dog Relation

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, makeSound(), which will be implemented by various classes.

- **Classes Dog, Cat, and Cow:** These classes implement the Animal interface, providing their own version of the makeSound() method.

- **Polymorphism in Action:** In the Main class, the reference myAnimal 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 myAnimal 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.

Real-World Example

Consider a payment system where different payment methods (e.g., CreditCard, PayPal, BankTransfer) implement a common Payment interface. The Payment interface might declare a method processPayment(), 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.

7.5 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
    }
}
```

7.6 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

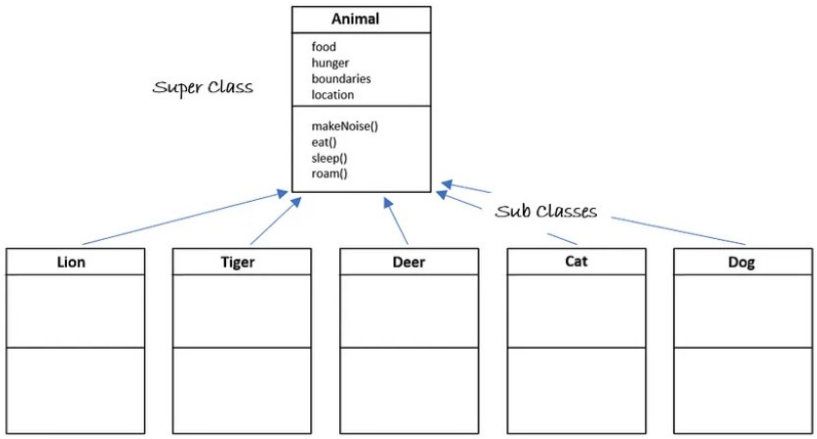


Fig 7.4. Interface and Abstract class implementation

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.

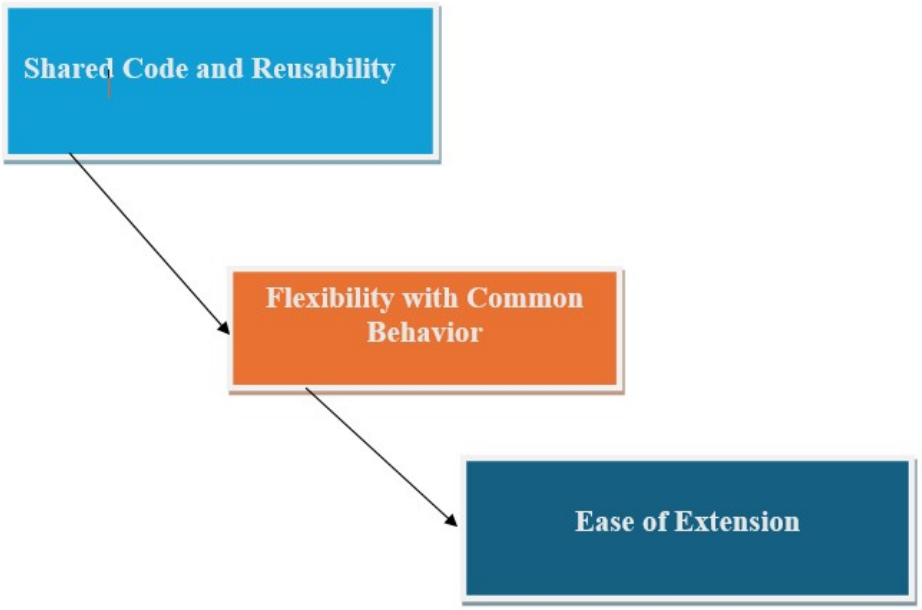


Fig 7.5 Benefits of Using Abstract Classes for Polymorphism

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.

7.8 Common Mistakes and Best Practices

8.1 Common Mistakes

- Confusing method overloading with overriding.

- Misusing polymorphism, leading to overly complex hierarchies.

8.2 Best Practices

- Favor composition over inheritance where possible.
- Keep class hierarchies shallow.
- Use @Override annotations to avoid accidental overloading.

7.9 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.

7.10 TECHNICAL TERMS

- Polymorphism
- Abstract Class
- Interface
- Compile Time
- Run Time
- Overriding

7.11 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?

7.12 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

AUTHOR: Dr. U.Surya Kameswari

LESSON- 8

INTERFACES

OBJECTIVES:

After going through this lesson, you will be able to

- Learn the syntax for defining an interface using the interface keyword.
- Understand the purpose of abstract methods, default methods, and static methods in interfaces.
- Compare interfaces and abstract classes in terms of usage, capabilities, and limitations.
- Understand how classes use the implements keyword to provide concrete implementations of interface methods.
- Understand how interfaces can extend other interfaces using the extends keyword.

STRUCTURE OF THE LESSION:

8.1 INTRODUCTION

In Java, interfaces play a crucial role in object-oriented programming by providing a mechanism to define a contract that classes must adhere to, ensuring abstraction and flexibility. Unlike abstract classes, which allow partial implementation with concrete methods, interfaces focus purely on the "what" rather than the "how," defining method signatures without implementation (except for default and static methods introduced in Java 8). Interfaces also differ from abstract classes in that they support multiple inheritance, allowing a class to implement multiple interfaces simultaneously, which is not possible with classes extending an abstract class. This distinction makes interfaces ideal for scenarios where a common behavior or functionality must be enforced across unrelated classes.

The chapter further explores the practical aspects of interfaces, starting with their definition using the interface keyword. It delves into how classes implement interfaces, ensuring all abstract methods are defined, and how interfaces can be extended to create more specific contracts. Accessing implementations through interface references highlights the power of polymorphism, enabling the use of different implementations interchangeably, promoting loose coupling and scalability. By understanding these topics, developers can harness interfaces to design modular, reusable, and maintainable code, a fundamental aspect of modern Java development.

8.2 ABSTRACT METHOD

An abstract method in Java is a method declared without an implementation (no method body). It acts as a placeholder that must be implemented by subclasses. Abstract methods are commonly used in abstract classes or interfaces to define behaviors that derived classes must provide.

An abstract method can be declared in an abstract class, which itself cannot be instantiated.

```
public abstract void display(); // Abstract method
```

Abstract methods must be implemented by subclasses (for abstract classes) or implementing classes (for interfaces). Abstract methods cannot be static, final, or private, as these modifiers prevent overriding, which defeats the purpose of abstraction.

An abstract method can be declared in an abstract class, which itself cannot be instantiated.

```
// Abstract class with an abstract method

public abstract class Shape {

   // Abstract method

   public abstract void draw();

   // Concrete method
```

```
    public void description() {
        System.out.println("This is a shape.");
    }
}
// Subclass providing implementation
public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle();  // Polymorphism
        shape.draw();               // Calls Circle's implementation
        shape.description();        // Calls concrete method
    }
}
```

Table 8.1 Abstract Methods vs. Concrete Methods

Feature	Abstract Methods	Concrete Methods
Definition	Declared without implementation (abstract).	Fully defined with a method body.
Location	Found in abstract classes or interfaces.	Found in regular or abstract classes.
Implementation Requirement	Must be implemented by subclasses or implementing classes.	Already implemented; can be overridden.
Modifiers	Cannot be static, final, or private.	Can use any valid method modifiers.

8.3 JAVA INTERFACE

An **interface** in Java is a reference type that defines a contract or blueprint for classes to implement. It specifies a set of abstract methods (without implementations) that implementing classes must provide. Interfaces are fundamental to achieving abstraction, polymorphism, and multiple inheritance in Java.

- **Creating an Interface**

Creating an interface in Java involves defining a blueprint for classes that will implement the interface. An interface specifies a set of methods that the implementing class must provide. Interfaces are declared using the interface keyword and can include abstract methods, constants, default methods, and static methods (Java 8+).

```
public interface Animal {
        // Abstract method
        void sound();
        // Constant (implicitly public static final)
        int LEGS = 4;
        // Default method
        default void sleep() {
           System.out.println("Sleeping...");
        }
        // Static method (Java 8+)
        static void info() {
           System.out.println("Animals are diverse.");
        }
   }
```

- Use the implements keyword.
- Provide concrete implementations for all abstract methods in the interface.

```
public class Dog implements Animal {
  @Override
        public void sound() {
           System.out.println("Bark");
        }
}
```

An interface reference to hold an object of a class that implements it. This promotes polymorphism.

```
public class Main {
```

```
    public static void main(String[] args) {
        Animal myDog = new Dog();  // Interface reference
        myDog.sound();            // Calls Dog's implementation
        myDog.sleep();            // Calls default method
        Animal.info();            // Calls static method
    }
}
```

8.4 INTERFACES Vs ABSTRACT CLASSES

Both interfaces and abstract classes are used to achieve abstraction in Java. However, they have distinct features, purposes, and use cases.

Table 8.2 Key Differences Between Interfaces and Abstract Classes

Aspect	Interface	Abstract Class
Purpose	Provides a contract that implementing classes must follow.	Serves as a base class with partial implementation for common functionality.
Keyword	Declared using the interface keyword.	Declared using the abstract keyword.
Method Implementation	All methods were abstract (until Java 8). Java 8+ allows default and static methods.	Can have both abstract and concrete methods.
Field Modifiers	Fields are implicitly public static final (constants).	Fields can have any access modifier and may include instance variables.
Constructor	Cannot have constructors.	Can have constructors to initialize fields or perform tasks.
Inheritance	Supports multiple inheritance (a class can implement multiple interfaces).	Does not support multiple inheritance (a class can extend only one abstract class).
Access Modifiers for Methods	Methods are implicitly public (cannot be private or protected).	Methods can have any access modifier (e.g., public, protected, private).
Static Context	Can include static methods (Java 8+) but no static blocks.	Can include static methods, static blocks, and static fields.
Use Case	Used to define a set of rules or behaviors across unrelated classes.	Used as a base for related classes with shared code and structure.

Similarities Between Interfaces and Abstract Classes

1. Both cannot be instantiated directly.
2. Both can enforce a set of methods for derived classes to implement.
3. Both support polymorphism and abstraction.

Use interfaces when designing a system with unrelated classes that share common behaviors or need multiple inheritance.

Use abstract classes when creating a base class for a group of related classes with shared code and structure.

8.5 IMPLEMENTING MULTIPLE INTERFACES

Java supports multiple inheritance through interfaces. A class can implement multiple interfaces by separating them with commas.

```
public interface Flyable {
    void fly();
}
public interface Swimmable {
    void swim();
}
public class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("Duck is flying");
    }
    @Override
    public void swim() {
        System.out.println("Duck is swimming");
    }
}
public class Main {
```

```
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.fly();
        duck.swim();
    }
}
```

Key Features When Implementing Interfaces

- **Overriding Methods:**
  - The class must use the @Override annotation when providing implementations for interface methods.
  - All interface methods are implicitly public, so implementations must also be public.
- **Polymorphism with Interfaces:**
  - Interfaces enable polymorphism, where multiple classes can be accessed through a single interface type.

```
public class Fish implements Swimmable {
    @Override
    public void swim() {
        System.out.println("Fish is swimming");
    }
}
Swimmable swimmable = new Fish();  // Polymorphic behavior
swimmable.swim();
```

- **Default and Static Methods (Java 8+):**
  - Default methods in an interface can be inherited without overriding unless customization is needed.
  - Static methods are accessed directly through the interface.

```
public interface Vehicle {
```

```
    default void start() {
        System.out.println("Vehicle starting...");
    }

    static void info() {
        System.out.println("Vehicle interface");
    }
}
public class Car implements Vehicle {}
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();          // Calls default method
        Vehicle.info();       // Calls static method
    }
}
```

Advantages of Implementing Interfaces

- **Achieves Multiple Inheritance:**
  - A class can implement multiple interfaces, allowing flexibility in design.
- **Promotes Polymorphism:**
  - Objects can be accessed through their interface type, enabling flexible code reuse.
- **Enforces a Contract:**
  - Ensures that all implementing classes follow the specified behavior.
- **Supports Loose Coupling:**
  - Code can interact with interface types rather than specific implementations.

Implementing interfaces in Java is a powerful way to enforce abstraction and flexibility. It allows you to design modular, reusable, and maintainable code by defining behaviors that multiple classes can share while retaining their individuality.

In Java, interfaces can extend other interfaces, similar to how classes can inherit from other classes. When an interface extends another interface, it inherits all the methods and constants of the parent interface. The child interface can also declare additional methods. This allows for the creation of more specific interfaces while maintaining a logical hierarchy.

8.6 EXTENDING MULTIPLE INTERFACES

An interface can extend multiple interfaces, allowing for a combination of behaviors. This is Java's way of supporting multiple inheritance for interfaces.

Example:

```
public interface Flyable {
    void fly();
}
public interface Swimmable {
    void swim();
}
public interface Amphibious extends Flyable, Swimmable {
    void liveOnLand();
}
```

❖ **Implementing Class:**

```
public class Duck implements Amphibious {
    @Override
    public void fly() {
        System.out.println("Duck is flying");
    }
    @Override
    public void swim() {
        System.out.println("Duck is swimming");
```

```
    }
    @Override
    public void liveOnLand() {
        System.out.println("Duck can live on land");
    }
}
public class Main {
    public static void main(String[] args) {
        Amphibious duck = new Duck();
        duck.fly();         // Output: Duck is flying
        duck.swim();        // Output: Duck is swimming
        duck.liveOnLand();  // Output: Duck can live on land
    }
}
```

- **Inheritance of Methods:**
  - A child interface inherits all the abstract methods and constants from its parent interfaces.
  - Implementing classes must implement all inherited methods.
- **Multiple Inheritance:**
  - An interface can extend multiple interfaces, which helps combine multiple behaviors.
  - Syntax:

```
public interface InterfaceC extends InterfaceA, InterfaceB {
    void methodC();
        }
```

- **Default and Static Methods:**
  - Default and static methods introduced in Java 8 can also be inherited or overridden when extending interfaces.

```
public interface ParentInterface {
    default void greet() {
        System.out.println("Hello from ParentInterface");
    }
}
public interface ChildInterface extends ParentInterface {
    @Override
    default void greet() {
        System.out.println("Hello from ChildInterface");
    }
}
```

- **Hierarchical Design:**
  - Extending interfaces promotes logical and organized design by defining broad behaviors in parent interfaces and more specific behaviors in child interfaces.

Advantages of Extending Interfaces

- **Reusability:**
  - Common functionality can be defined in the parent interface and reused by multiple child interfaces.
- **Scalability:**
  - New features or behaviors can be added without modifying existing interfaces or classes.
- **Multiple Inheritance:**
  - Combines capabilities from multiple parent interfaces, overcoming the single inheritance limitation of classes.

Extending interfaces in Java provides a way to build hierarchies and modularize behavior. It allows for code reusability, scalability, and logical separation of responsibilities, making it a powerful tool in designing maintainable and extensible applications.

8.7 SUMMARY

In conclusion, interfaces in Java play a critical role in defining contracts for classes to follow, allowing for a high level of abstraction and flexibility in object-oriented design. Unlike

abstract classes, interfaces cannot provide method implementations (except for default and static methods in Java 8 and beyond) and are primarily used to define a set of behaviors that can be implemented by any class, regardless of its place in the class hierarchy. The key advantage of interfaces over abstract classes lies in their ability to support multiple inheritance, allowing a class to implement multiple interfaces, thus providing a way to combine different behaviors. This contrasts with abstract classes, which are limited to single inheritance and are more suited for situations where a common base implementation is needed.

When implementing interfaces, a class must provide concrete implementations for all the abstract methods declared in the interface, ensuring that the contract defined by the interface is fully realized. Accessing implementations through interface references allows for polymorphic behavior, where different objects can be treated uniformly as instances of the same interface type. Additionally, interfaces can be extended, enabling the creation of more specific interfaces by inheriting the methods of parent interfaces and adding new ones. This hierarchical structure promotes modular, reusable, and scalable code, making interfaces a fundamental concept for creating flexible and maintainable software in Java.

8.8 TECHNICAL TERMS

- Interface
- Abstract Class
- Abstract Method
- Contract
- Polymorphism
- Multiple Inheritance
- Implementing an Interface
- implements Keyword
- Method Implementation
- Interface Reference
- Dynamic Binding
- Default Method
- Static Method
- extends Keyword
- Extending Interfaces

8.9 SELF ASSESSMENT QUESTIONS

Essay questions:

1. What are the Key Differences Between Interfaces and Abstract Classes in Java? Discuss When to Use Each One with Suitable Examples.
2. How Are Interfaces Defined in Java? Discuss the Structure of an Interface and the Types of Methods it Can Contain. Provide Examples of Different Types of Methods within an Interface.
3. What is the Process of Implementing an Interface in Java? Explain the Steps Involved and the Rules That Must Be Followed. Include Examples to Illustrate the Implementation Process.
4. Explain How Accessing Implementations Through Interface References Works in Java. Discuss the Concept of Polymorphism and How It Enables Flexibility in Code Design. Provide Practical Examples.

Short Answer Questions:

1. How do you define an interface in Java?\
2. What is the role of the implements keyword in Java?
3. Can an interface reference be used to access objects of multiple classes? Explain.
4. What does it mean to extend an interface in Java?

8.10 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

AUTHOR: **Dr. U. Surya Kameswari**

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 interfac
- 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 OF THE LESSION:

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’.

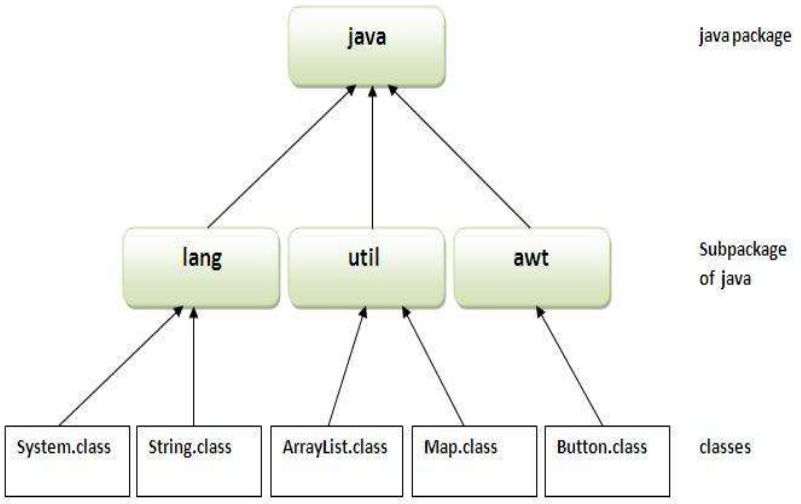


Figure 9.1 built in packages

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.

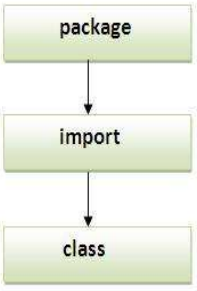


Figure 9.2 package accessing hierarchy

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.
    }
}
```

Most Restrictive ← → Least Restrictive

Access Modifiers ->	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

Access Modifiers and Packages

Packages play a significant role in controlling access to classes and their members:

1. **Public: Classes or members declared as public are accessible from any other class, even from different packages.**
2. **Protected: protected members are accessible within the same package or by subclasses.**
3. **Default (Package-Private): If no access modifier is provided, the member is only accessible within the same package.**
4. **Private: private members are only accessible within the same class.**

Benefits of Using Packages

1. **Namespace Management**: Packages prevent class name conflicts by grouping related classes together.
2. **Access Control**: Packages allow control over the accessibility of classes, interfaces, and methods using access modifiers.
3. **Code Organization**: Large programs are easier to manage when organized into packages, improving readability and maintainability.
4. **Reusability**: Classes and methods from other packages can be reused, promoting code reuse.
5. **Ease of Maintenance**: Modular design enables faster debugging and easier updates.

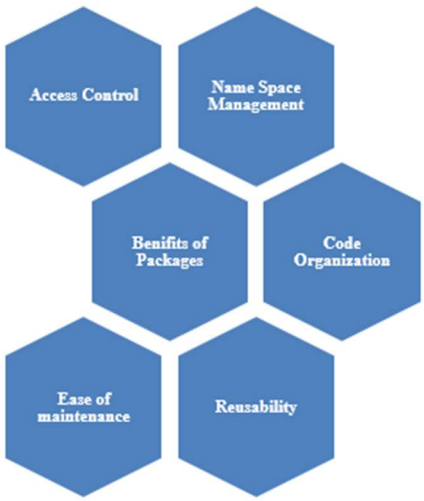


Figure 9.4 Benefits of Using Packages

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 . Illustate 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

AUTHOR: **Dr. Vasantha Rudramalla**

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 OF THE LESSION:

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 if 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

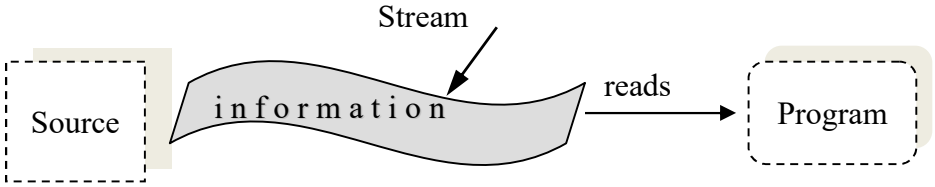


Figure 10.1 read stream

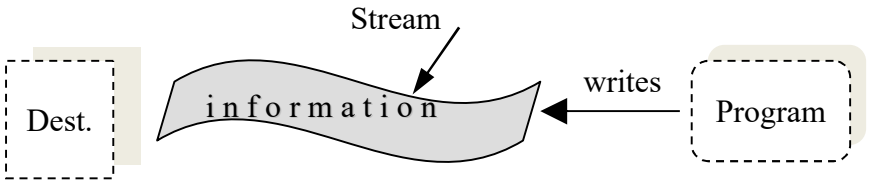


Figure 10.2 Write stream

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.

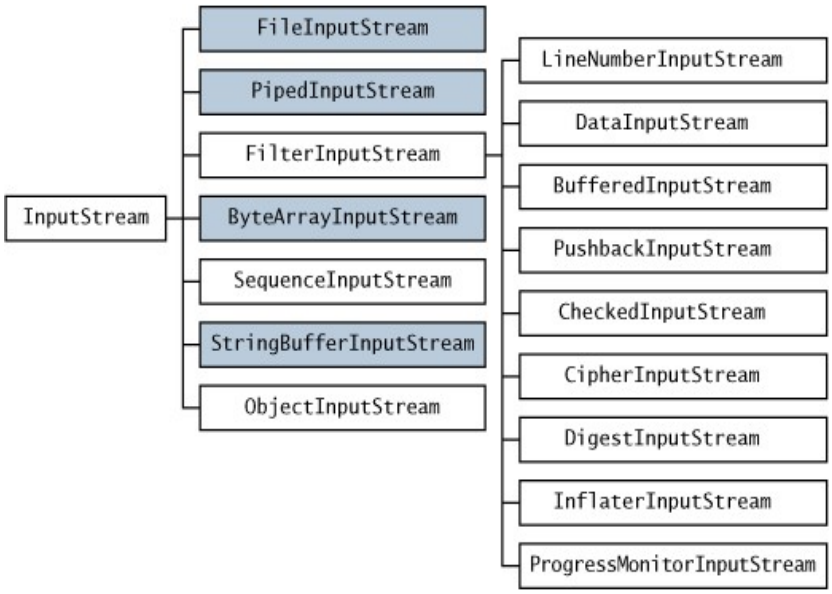


Figure 10.3 input stream class hierarchy

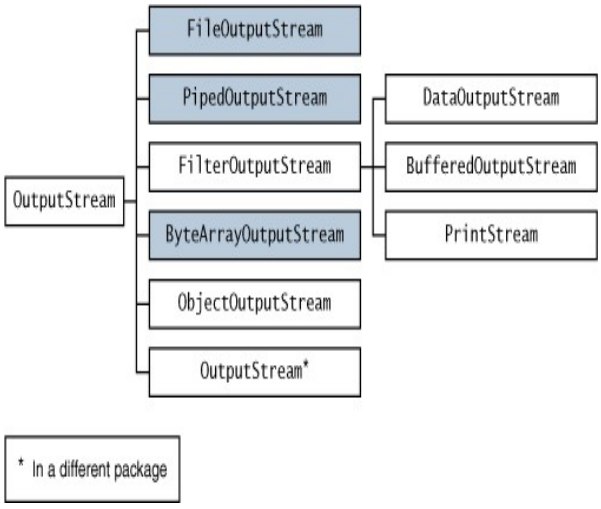


Figure 10. 4 outputstream class hierarchy

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.

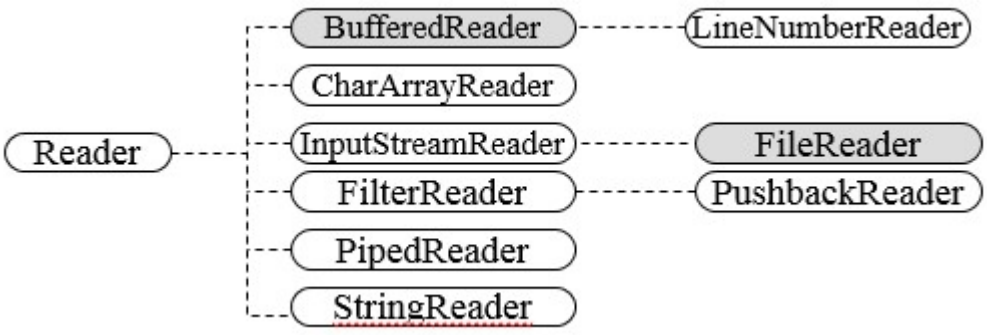


Figure 10.6 Readerr class stream hierarchy

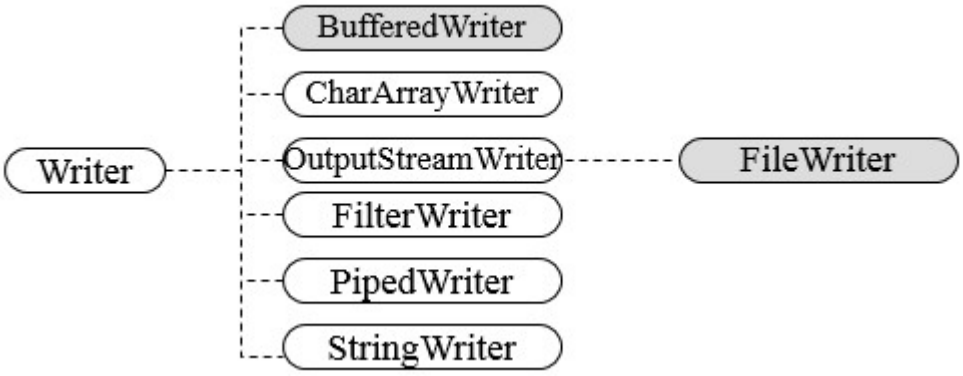


Figure 10.6 Writer class stream hierarchy

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.

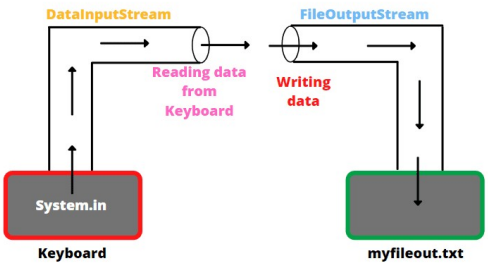


Figure 10.7 : Creating a text file

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.

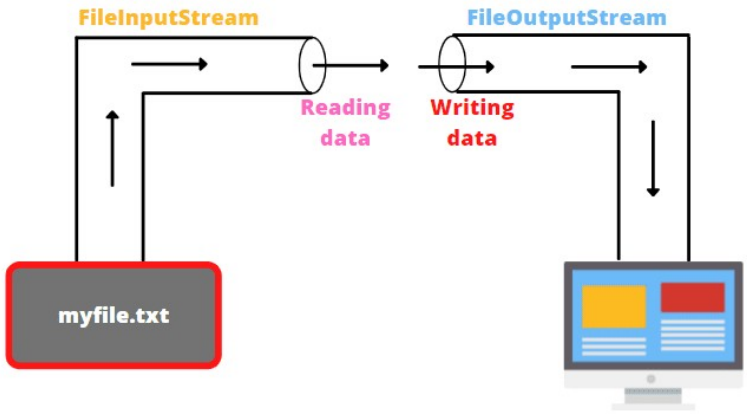


Figure 10.8 : Reading data from a text file

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.

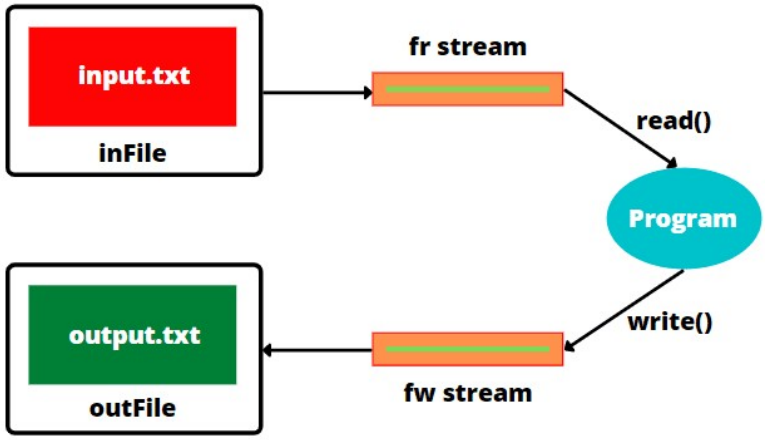


Figure 10.9 Reading from and writing to files

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 BINARY INPUT AND OUTPUT IN JAVA

Binary Input and Output (I/O) in Java refers to reading and writing data in binary format (i.e., as raw bytes) rather than the typical text format. Binary I/O is used for handling non-text data like images, audio files, videos, and other types of binary data.

10.6.1 Binary Input in Java

Binary input involves reading data as raw bytes from a file or other data source. The InputStream class and its subclasses (such as FileInputStream) are commonly used for reading binary data.

Example: Reading Binary Data from a File

```
import java.io.FileInputStream;

import java.io.IOException;

public class BinaryInputExample {

   public static void main(String[] args) {

      try (FileInputStream fis = new FileInputStream("example.bin")) {

         int data;

         while ((data = fis.read()) != -1) {

            System.out.print((char) data); // Reads byte-by-byte and prints

         }

      } catch (IOException e) {

         e.printStackTrace();

      }

   }

}
```

In this example:

- FileInputStream is used to read binary data from a file.
- The read() method reads one byte at a time.
- -1 indicates the end of the file.

10.6.2 Binary Output in Java

Binary output involves writing data as raw bytes to a file or another destination. The OutputStream class and its subclasses (such as FileOutputStream) are used to write binary data.

Example: Writing Binary Data to a File

```
import java.io.FileOutputStream;

import java.io.IOException;

public class BinaryOutputExample {

   public static void main(String[] args) {

      try (FileOutputStream fos = new FileOutputStream("example.bin")) {

         String data = "Hello, Binary World!";

         byte[] byteArray = data.getBytes();

         fos.write(byteArray); // Writes binary data to the file

      } catch (IOException e) {

         e.printStackTrace();

      }

   }

}
```

In this example:

- FileOutputStream is used to write binary data to a file.
- The getBytes() method converts the String into a byte array.
- The write() method writes the byte array to the file.

10.6.3 Handling Binary Data with Streams

Java provides different classes to handle binary I/O:

1. **FileInputStream**: Reads data from a file as bytes.
2. **FileOutputStream**: Writes data to a file as bytes.
3. **BufferedInputStream**: Provides efficient reading of binary data by buffering input.

4. **BufferedOutputStream**: Provides efficient writing of binary data by buffering output.

10.6.4 Advantages of Binary I/O

- **Efficiency**: Binary I/O is generally faster and more efficient than text-based I/O because it doesn't involve character encoding and decoding.
- **Precision**: It allows precise representation of data (e.g., storing a 4-byte int exactly as it is in memory, instead of converting it to a string).
- **Handling Complex Data**: Binary I/O is essential for reading and writing non-text files like images, audio, and video files.

Binary Input and Output in Java is used for handling raw data such as images, audio, and other non-text formats. Using InputStream and OutputStream classes, Java allows efficient reading and writing of binary data, providing more control over the data format compared to text based I/O.

10.7 RANDOM ACCESS OPERATIONS

Random access refers to the ability to access elements at any position in a file or data structure directly, without needing to read through the entire file or structure sequentially. In the context of file I/O in Java, random access allows for reading or writing to any part of a file efficiently, without needing to process it from the beginning.

Java provides the RandomAccessFile class, which allows for both reading and writing to files at any position, making it ideal for tasks where you need to access data at random positions within a file, such as in databases, media files, or other structured data.

❖ **Key Features of Random Access in Java**

1. File Pointer: RandomAccessFile maintains a pointer (or cursor) that determines the current position within the file. You can move this pointer to any byte in the file using seek() and perform operations like reading and writing at that position.
2. Efficient Access: Unlike sequential access where data is processed in a linear fashion, random access allows for efficient manipulation of data by directly moving the file pointer to the desired position.
3. Read and Write Operations: RandomAccessFile supports both read and write operations, allowing for flexible file manipulation.

❖ **Creating a RandomAccessFile Object**

To perform random access operations, you need to create a RandomAccessFile object, specifying the file name and the mode (either "r" for read or "rw" for read-write).

```
import java.io.RandomAccessFile;
```

```
import java.io.IOException;

public class RandomAccessExample {
    public static void main(String[] args) {
        try {
            // Creating a RandomAccessFile object in read-write mode
            RandomAccessFile file = new RandomAccessFile("example.txt", "rw");
             // Perform operations on the file here
            file.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In this example:

- RandomAccessFile is created in read-write mode ("rw").
- The file is now open for random access operations.

❖ **Seeking a Position in the File**

You can move the file pointer to any position in the file using the seek(long pos) method. The parameter pos specifies the byte position in the file (starting from 0).

Example: Moving the Pointer to a Specific Position

```
import java.io.RandomAccessFile;

import java.io.IOException;

public class SeekExample {
    public static void main(String[] args) {
        try {
            RandomAccessFile file = new RandomAccessFile("example.txt", "rw");
```

```
            // Moving the pointer to the 5th byte
            file.seek(5);
            // Read data from this position
            int data = file.read();
            System.out.println("Data at position 5: " + (char) data);
              file.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In this example:

- seek(5) moves the file pointer to the 5th byte in the file.
- read() reads the byte from that position.

❖ **Reading and Writing Data**

After moving the pointer using seek(), you can perform read and write operations just like with regular file I/O.

Example: Writing Data at a Specific Position

```
import java.io.RandomAccessFile;
import java.io.IOException;
public class RandomWriteExample {
    public static void main(String[] args) {
        try {
            RandomAccessFile file = new RandomAccessFile("example.txt", "rw");
             // Move the pointer to the 10th byte and write data
            file.seek(10);
            file.writeBytes("Hello, World!");
```

```
            file.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In this example:

- seek(10) moves the pointer to the 10th byte.
- writeBytes("Hello, World!") writes the string starting from that position.

❖ **Reading Data at Random Positions**

You can read data from any position by moving the pointer and then calling the appropriate read methods (readByte(), readChar(), readInt(), etc.).

Example: Reading Integers at Random Positions

```
import java.io.RandomAccessFile;
import java.io.IOException;
public class RandomReadExample {
    public static void main(String[] args) {
        try {
            RandomAccessFile file = new RandomAccessFile("example.dat", "r");
              // Moving the pointer to the 4th byte and reading an integer
            file.seek(4);
            int data = file.readInt(); // Reads 4 bytes (an integer)
            System.out.println("Integer at position 4: " + data);
             file.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In this example:

- seek(4) moves the pointer to the 4th byte.
- readInt() reads a 4-byte integer from that position.

❖ **Modifying File Content**

You can overwrite data at any position in the file. By using the seek() method, you can move the file pointer to a specific byte and modify the content there. However, keep in mind that writing to a position before the current size of the file will truncate the file content, which might cause data loss.

❖ **Advantages of Random Access**

- Efficient File Handling: Random access allows you to read or write at any position in a file without needing to load the entire file into memory.
- Direct Data Manipulation: You can efficiently modify specific parts of the file, such as in databases or media files, where access to particular records or frames is necessary.
- Improved Performance: For large files, random access minimizes the need for sequential reads and writes, making it a faster approach for certain types of file manipulation.

❖ **Limitations of Random Access**

- Not Suitable for All Data Types: Random access is ideal for binary files or structured data but may not be as useful for text files or other data types that require sequential reading and interpretation.
- Complexity: Managing file pointers and keeping track of positions in large files can introduce complexity in your program logic.

10.8 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.

10.9 TECHNICAL TERMS

Stream, InputOutput, Byte stream, Character stream, File

10.10 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?

10.11 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

AUTHOR: **Dr. Vasantha Rudramalla**

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 OF THE LESSION:

11.1 INTRODUCTION

Errors and exceptions are integral concepts in Java programming, representing situations where the normal flow of a program is disrupted. Errors typically refer to serious issues that are beyond the program's control, such as hardware failure or memory exhaustion, and are categorized under the Error class. Exceptions, on the other hand, are conditions that arise during runtime and can potentially be handled by the program to maintain smooth execution. Java provides a robust mechanism to handle such scenarios through its exception hierarchy, which includes checked exceptions, unchecked exceptions, and errors, all derived from the Throwable superclass.

Exception handling is a fundamental aspect of Java, allowing developers to gracefully manage runtime anomalies. Using constructs like try, catch, and finally, programs can

anticipate and recover from exceptions without abruptly terminating. The throw and throws clauses further refine exception handling. The throw clause is used to explicitly raise an exception within a method, while the throws clause declares the exceptions that a method may propagate, ensuring proper communication between different program components. Together, these mechanisms provide a structured approach to creating reliable, fault-tolerant Java applications.

11.2 ERRORS IN JAVA PROGRAM

Errors in Java can be broadly classified into several categories.

11.2.1. Syntax Errors:

Syntax errors are errors in the structure of our code, usually detected at compile time. These types 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
 }
}
```

11.2.2. Runtime Errors:

Runtime errors are errors that occur while the program is running, leading to abnormal termination. These types 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
            }
    }
```

11.2.3. 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
        }
    }
}
```

11.2.4. 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
    }
}
```

11.3 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 another place in the program.

The exception object stores information about the nature of the problem. For example, due to network problems or classes not found etc.

A Java Exception is an object that describes the exception that occurs in a program.

When an exception 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***.**

11.3.1 Various Categories of Exceptions

1123.1.1. 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 RuntimeException 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 IOException, SQLException, and other exceptions are examples of checked exceptions.

11.3.1.2. 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 RuntimeException class are known as Unchecked Exceptions." Examples of exceptions include the ArithmeticException and the NullPointerException.

11.3.1.3 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.

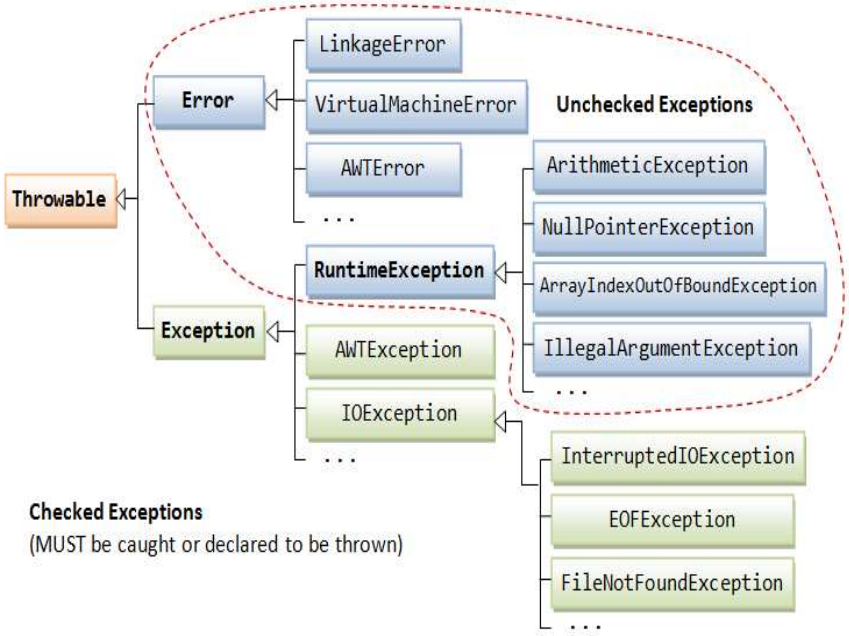


Figure 11.1 Types of Exceptions

Differences between checked and unchecked exceptions

Checked Exceptions	Unchecked Exceptions
• 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 RuntimeException and Error are known as checked exceptions • e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.	• represent defects in the program (bugs) • checked at run time • Unchecked exception are ignored at compile time. • The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

11.4 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.

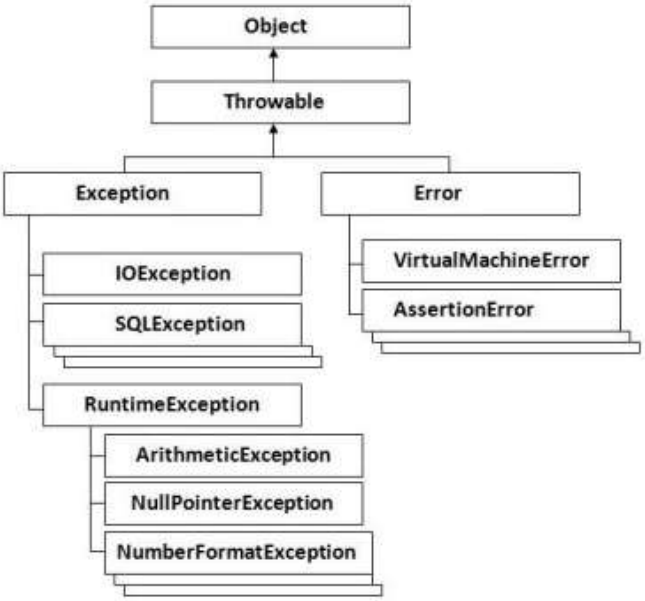


Figure 11.2 Exception hierarchy

11.4.1 Built in Exceptions:

List of Java Unchecked exceptions under RuntimeException

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.
StringIndexOutOfBounds	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.

11.5 EXCEPTION HANDLING

Exception Handling is the mechanism to handle runtime malfunctions. We need to handle such exceptions to prevent abrupt termination of the 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. To accomplish this, we first try 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

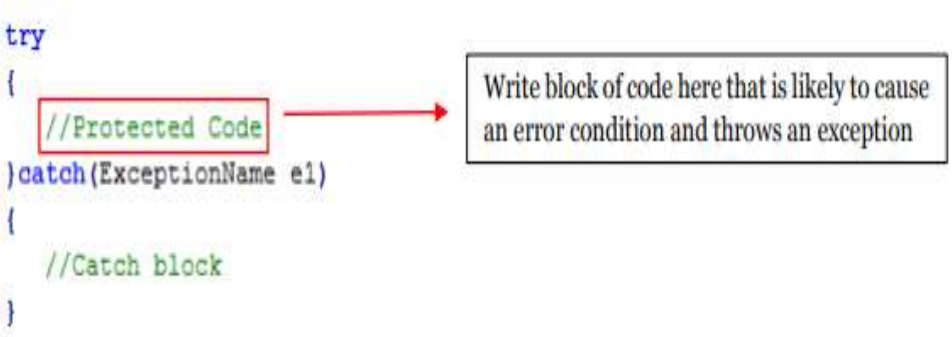
11.5.1 try – catch block

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

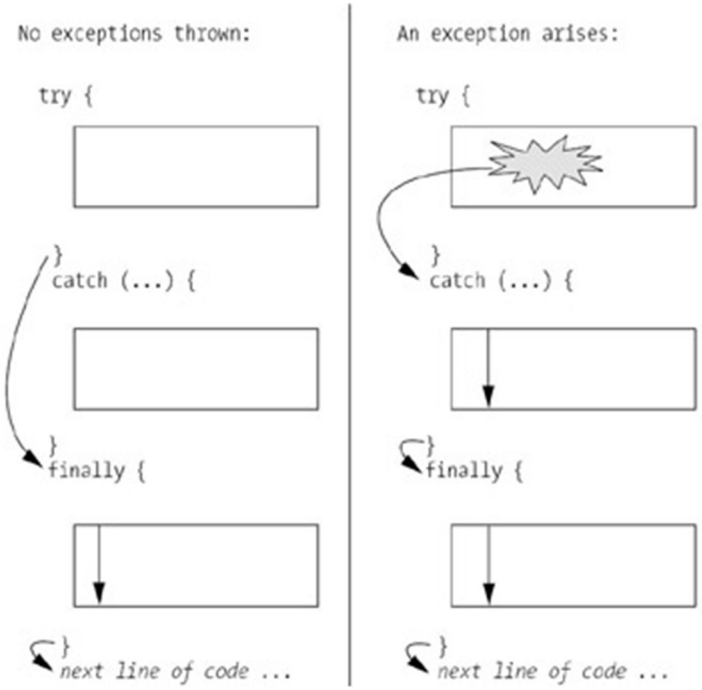
Syntax:



Fgure 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.



Fgure 11.4 try catch finally block

11.5.1.1 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**,** **t**he program throws an `ArithmeticException` and terminates immediately when the exception occurs.

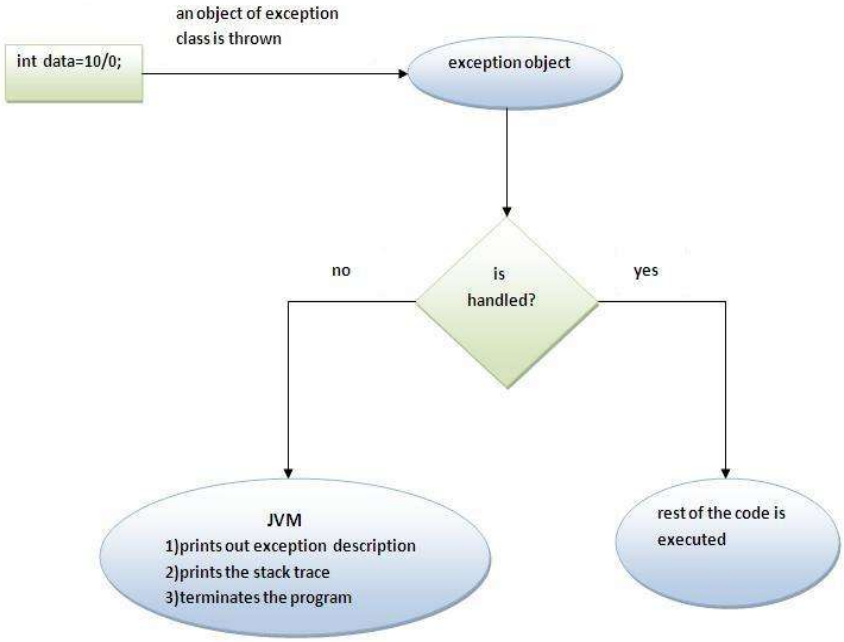


Figure 11.5 without exception handling

11.5.1.2 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**, t**he 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.

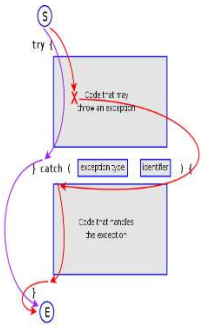


Fig 11.6 try-catch block

11.5.2 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.

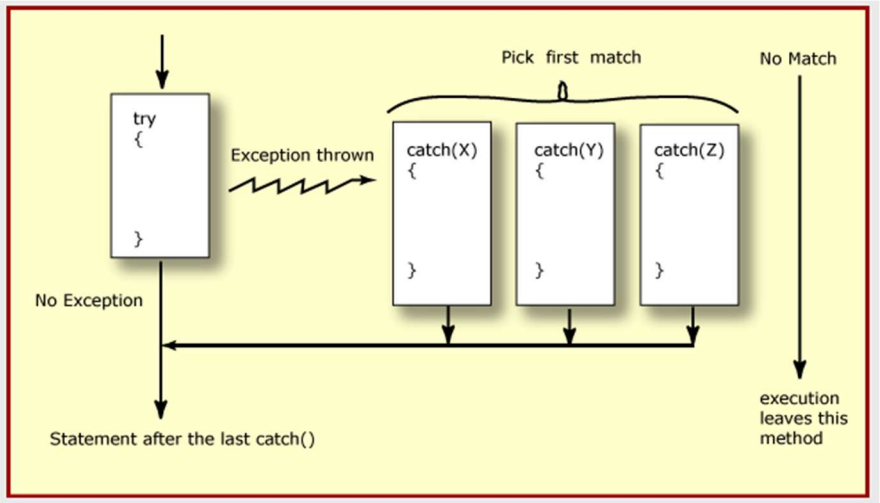


Figure 11.7 multiple catch blocks

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.

11.5.5.5 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.

11.5.6 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.

11.5.6.1 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
}
```

11.5.6.2 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.

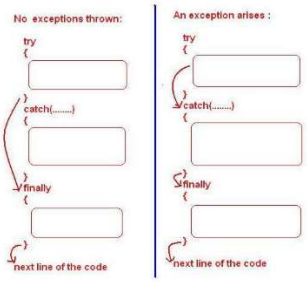


Figure 11.8 Finally block

Example

```
public class FinallyBlockExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);  // This will throw an ArithmeticException
            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.

11.6 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.

11.7 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.

11.8 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.

11.9 TECHNICAL TERMS

Error, exception, try, catch, throw , throws

11.10 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?

11.11 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

AUTHOR: **Dr. Vasantha Rudramalla**

LESSON- 12

BUILT-IN & OWN EXCEPTIONS

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the built-in exceptions with detail
- Learn the difference between checked and unchecked exceptions.
- Learn how to create your own exceptions with detail

STRUCTURE OF THE LESSION:

12.1 INTRODUCTION

Exception handling is a critical feature in Java that ensures the smooth execution of programs, even in the presence of errors or unexpected conditions. Exception handling is a cornerstone of Java programming, enabling developers to address unexpected conditions and runtime errors effectively. Java provides a robust exception hierarchy, including built-in exceptions such as IOException, NullPointerException, and ArithmeticException, to handle common errors. These exceptions simplify error detection and recovery but may not always align with specific application needs. To address unique business requirements and improve clarity, Java allows developers to create their own custom exception classes. Custom exceptions enhance code readability by providing domain-specific error representations, making it easier to debug and maintain applications. This chapter delves into Java's built-in exceptions, exploring their usage and limitations, while also guiding readers on designing custom exception subclasses. Best practices for exception handling are discussed to ensure clean, efficient, and reliable code that is easy to manage and extend.

12.2 KEY FEATURES OF EXCEPTION HANDLING

Its importance can be summarized through the following key points:

1. Ensures Program Stability

- Exceptions allow programs to handle errors gracefully without abruptly crashing.
- By catching and handling exceptions, developers can ensure that the program continues to function for unaffected operations, providing a seamless user experience.

2. Improves Debugging and Maintenance

- Java provides detailed exception messages and stack traces, helping developers pinpoint the cause and location of errors.
- Proper exception handling structures the code to isolate and address specific error scenarios, making debugging and maintenance more manageable.

3. Encourages Robust Code Design

- With exception handling, developers anticipate potential issues and write code that accounts for edge cases.
- It enforces defensive programming practices, leading to more resilient and reliable applications.

4. Facilitates Error Recovery

- Exception handling allows programs to recover from errors instead of terminating. For example:
  - Retrying an operation in case of network failure.
  - Using alternative resources when a file is unavailable.

5. Enables Separation of Concerns

- The use of try, catch, and finally blocks separates error-handling logic from regular business logic.
- This separation improves code readability and modularity.

6. Supports Resource Management

- Exception handling is essential for managing resources like file streams, database connections, and network sockets.
- The try-with-resources feature in Java ensures that resources are automatically closed, even if an exception occurs, preventing resource leaks.

7. Comprehensive Error Communication

- Custom exceptions enable developers to create meaningful, context-specific error messages.
- This improves communication between different parts of the program and helps developers understand and resolve issues more effectively.

8. Fosters Standardized Error Handling

- Java's structured hierarchy ensures consistency in how errors are defined and handled.
- By adhering to Java's conventions, developers create predictable and understandable error-handling mechanisms.

Example

Consider a file-reading operation:

```
try {
   BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
   String line = reader.readLine();
   System.out.println(line);
   reader.close();
} catch (FileNotFoundException e) {
   System.err.println("File not found: " + e.getMessage());
} catch (IOException e) {
   System.err.println("Error reading file: " + e.getMessage()); }
```

This example highlights how exception handling ensures the program can handle missing files or read errors without crashing. Exception: handling Java is essential for building robust, user-friendly, and maintainable applications. It equips developers with tools to anticipate, manage, and recover from runtime errors, ensuring applications run reliably in diverse environments.

12.3 COMPARISION OF CHECKED AND UNCHECKED EXCEPTION

Aspect	Checked Exceptions	Unchecked Exceptions
Definition	Exceptions that must be either handled using a try-catch block or declared in the throws clause of a method.	Exceptions that are not required to be explicitly handled or declared in the throws clause.
Examples	IOException, SQL Exception, ClassNotFoundException	NullPointer Exception, Array IndexOutOfBoundsException, ArithmeticException
Category	Subclasses of the Exception class, excluding RuntimeException.	Subclasses of the RuntimeException class.
Compile-Time Enforcement	Checked at compile time. The compiler ensures that these exceptions are properly handled.	Not checked at compile time. Developers are responsible for handling these exceptions.
Nature of Errors	Represent recoverable errors that can occur in normal program operation.	Represent programming errors or bugs, such as invalid arguments or logical mistakes.
Use Cases	Used for predictable and recoverable situations, such as file not found or invalid user input.	Used for unexpected and usually irrecoverable situations, such as null pointer dereferences or division by zero.
Declaration Requirement	Must be declared in the method signature using the throws keyword if not handled within the method.	No declaration in the method signature is required.
Impact on Code	Requires more verbose handling due to enforced error management.	Simplifies code as handling is optional but relies on careful coding to avoid runtime crashes.
Best Practices	Use when the program can and should recover from the exception.	Use for errors caused by bugs or situations unlikely to be recovered during runtime.

Example of Checked Exception

```
import java.io.*;

public void readFile(String filePath) throws IOException {

   BufferedReader reader = new BufferedReader(new FileReader(filePath));

   System.out.println(reader.readLine());

   reader.close();

}
```

In this example, IOException is a checked exception, and the method must declare it in the throws clause or handle it within a try-catch block.

Example of Unchecked Exception

```
public int divide(int a, int b) {

   return a / b; // May throw ArithmeticException if b is 0

}
```

Here, ArithmeticException is an unchecked exception. It is not required to be handled explicitly but will cause a runtime crash if not managed.

- Checked Exceptions: Suitable for recoverable errors; enforce disciplined error handling at compile time.
- Unchecked Exceptions: Used for programmer errors or unexpected scenarios; offer flexibility but require diligence to avoid runtime failures.

12.4 CUSTOM EXCEPTIONS

Custom exceptions in Java are essential for building robust and maintainable applications. They allow developers to define application-specific error conditions, improving clarity, usability, and debugging. Here's a detailed explanation of their necessity, focusing on addressing specific domain requirements and enhancing code readability and debugging:

1. Addressing Specific Domain Requirements

Custom exceptions are tailored to the unique needs of an application or domain, enabling precise handling of domain-specific issues. Unlike built-in exceptions, which are generic, custom exceptions convey the exact nature of a problem in the context of the application.

- Clearer Problem Identification: Custom exceptions make it immediately apparent what went wrong within a specific domain. For example, in a banking application:

- Instead of throwing a generic Exception, a LowBalanceException explicitly signals an insufficient funds issue.

- Improved Error Handling Logic: They enable developers to implement targeted handling strategies for specific scenarios. For instance:
  - A PaymentFailedException might trigger a retry mechanism.
  - A UserNotAuthorizedException could redirect the user to a login page.
- Domain-Specific Workflows: Custom exceptions help enforce rules and workflows in the domain model. For example:

```
public class InvalidAgeException extends Exception {
   public InvalidAgeException(String message) {
      super(message);
   }
}
public void registerUser(int age) throws InvalidAgeException {
   if (age < 18) {
      throw new InvalidAgeException("Age must be 18 or older.");
   }
}
```

This makes the business rule (minimum age requirement) explicit in the code.

2. Enhancing Code Readability and Debugging

Custom exceptions make code easier to understand and debug by clearly communicating the cause of an error.

- Meaningful Exception Names: Custom exceptions with descriptive names, such as InvalidTransactionException or ResourceNotAvailableException, convey the exact issue without requiring developers to read error messages or dig into code.
- Separation of Concerns: By introducing custom exceptions, error-handling logic becomes more modular and focused on specific scenarios. This separation enhances readability and maintainability.

```
try {
   withdraw(amount);
```

```
} catch (LowBalanceException e) {
    System.out.println("Withdrawal failed: " + e.getMessage());
}
```

- Rich Context for Debugging: Custom exceptions allow developers to include additional details, such as error codes or metadata, providing a richer context for debugging.

```
public class OrderProcessingException extends Exception {
    private int orderId;
    public OrderProcessingException(String message, int orderId) {
        super(message);
        this.orderId = orderId;
    }
    public int getOrderId() {
        return orderId;
    }
}
```

- Enhanced Stack Traces: Custom exceptions make stack traces easier to interpret because they are directly tied to application-specific logic. A trace that includes PaymentGatewayException is far more informative than one with a generic Exception.
- Consistent Error Communication: Using custom exceptions enforces consistent patterns for error reporting across the application, making logs and debugging outputs more organized.

Benefits in Practice

1. Domain-Specific Clarity: When a ProductOutOfStockException is thrown, it clearly indicates the problem compared to a generic RuntimeException.
2. Precise Handling: Custom exceptions allow focused error-handling strategies without needing to parse error messages or use catch-all blocks.
3. Improved User Experience: End-users can receive clear and actionable error messages derived from custom exceptions.

12. 5 STEPS TO CREATE CUSTOM EXCEPTION

Creating a custom exception in Java involves defining a new class that extends an existing exception class. Follow these steps to create and use a custom exception:

1. Decide on the Type of Exception

- Checked Exception: Extend the Exception class if the error condition must be explicitly handled or declared (throws clause).
- Unchecked Exception: Extend the RuntimeException class if the error represents a programming error or is optional to handle.

2. Define the Custom Exception Class

- Extend the appropriate base exception class (Exception or RuntimeException).
- Add constructors to allow various ways of initializing the exception.

3. Implement Constructors

Include constructors that:

1. Default Constructor: Initializes the exception without any details.
2. Message Constructor: Accepts a custom error message for descriptive errors.
3. Message and Cause Constructor: Provides both a custom message and the underlying cause of the error.

Example Code:

```
public class CustomException extends Exception {
   // Default constructor
   public CustomException() {
      super("Default error message");
   }
   // Constructor with a custom message
   public CustomException(String message) {
      super(message);
   }
   // Constructor with a custom message and cause
   public CustomException(String message, Throwable cause) {
      super(message, cause);
```

```
    }
}
```

4. Optionally Add Custom Fields

If additional context is needed, include custom fields with getter methods for more detailed error information.

Example Code:

```
public class UserNotFoundException extends Exception {
    private int userId;
    public UserNotFoundException(String message, int userId) {
        super(message);
        this.userId = userId;
    }
    public int getUserId() {
        return userId;
    }
}
```

5. Use the Custom Exception

Throw the custom exception in your application logic when the specific error condition occurs.

Example Code:

```
public void findUser(int userId) throws UserNotFoundException {
    if (userId != 12345) { // Simulating a condition
        throw new UserNotFoundException("User not found with ID: " + userId, userId);
    }
    System.out.println("User found!");
}
```

6. Handle the Custom Exception

Handle the exception using a try-catch block or propagate it further using the throws clause.

Example Code:

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

```
            findUser(999); // Pass an invalid ID
        } catch (UserNotFoundException e) {
            System.err.println("Error: " + e.getMessage());
            System.err.println("User ID: " + e.getUserId());
        }
    }
```

❖ **Test the Exception**

Ensure the custom exception behaves as expected:

- Verify it is thrown under the correct conditions.
- Confirm that the error message and context are accurate.

12.6 REAL-TIME APPLICATIONS OF CUSTOM EXCEPTIONS

Custom exceptions are widely used in real-world applications to handle domain-specific errors, improve clarity, and create more robust error-handling mechanisms. Here are examples demonstrating practical use cases:

1. E-commerce Application: Handling Insufficient Stock

In an e-commerce application, when a customer attempts to purchase more items than are available, a InsufficientStockException can be thrown.

Code Example

```
// Custom exception for insufficient stock
public class InsufficientStockException extends Exception {
    public InsufficientStockException(String message) {
        super(message);
    }
}

// Inventory class
public class Inventory {
    private int stock;
```

```
    public Inventory(int stock) {
        this.stock = stock;
    }

    public void purchaseItem(int quantity) throws InsufficientStockException {
        if (quantity > stock) {
            throw new InsufficientStockException("Insufficient stock. Available: " + stock);
        }
        stock -= quantity;
        System.out.println("Purchase successful! Remaining stock: " + stock);
    }
}

// Main class
public class ECommerceApp {
    public static void main(String[] args) {
        Inventory inventory = new Inventory(10);

        try {
            inventory.purchaseItem(12); // Attempting to purchase more than available
        } catch (InsufficientStockException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

2. Banking Application: Handling Insufficient Balance

In a banking application, a InsufficientBalanceException can be used to handle scenarios where a user tries to withdraw more money than their account balance.

Code Example

```
// Custom exception for insufficient balance
public class InsufficientBalanceException extends RuntimeException {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}

// BankAccount class
public class BankAccount {
    private double balance;
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
    public void withdraw(double amount) {
        if (amount > balance) {
            throw new InsufficientBalanceException("Insufficient balance. Available: " + balance);
        }
        balance -= amount;
        System.out.println("Withdrawal successful! Remaining balance: " + balance);
    }
}
// Main class
public class BankingApp {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(500.0);
```

```
        try {
            account.withdraw(600.0); // Attempting to withdraw more than the balance
        } catch (InsufficientBalanceException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

3. Student Management System: Invalid Age Entry

In an education system, a InvalidAgeException can be used to ensure that students meet the minimum age requirement for admission.

Code Example

```
// Custom exception for invalid age
public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
// Student class
public class Student {
    public void registerStudent(String name, int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older for registration.");
        }
        System.out.println("Student registered successfully: " + name);
    }
}
```

```
// Main class
public class StudentManagementApp {
    public static void main(String[] args) {
        Student student = new Student();


        try {
            student.registerStudent("John Doe", 16); // Invalid age
        } catch (InvalidAgeException e) {
            System.err.println("Registration error: " + e.getMessage());
        }
    }
}
```

4. Online Booking System: Invalid Date for Reservation

In an online booking system, a InvalidBookingDateException can validate the reservation date to ensure it is not in the past.

Code Example

```
import java.time.LocalDate;
// Custom exception for invalid booking date
public class InvalidBookingDateException extends Exception {
    public InvalidBookingDateException(String message) {
        super(message);
    }
}
// Booking class
public class Booking {
    public void makeReservation(LocalDate bookingDate) throws
InvalidBookingDateException {
        if (bookingDate.isBefore(LocalDate.now())) {
```

```
            throw new InvalidBookingDateException("Booking date cannot be in the past.");
        }
        System.out.println("Reservation successful for date: " + bookingDate);
    }
}
// Main class
public class BookingApp {
    public static void main(String[] args) {
        Booking booking = new Booking();
        try {
            booking.makeReservation(LocalDate.of(2023, 11, 20)); // Past date
        } catch (InvalidBookingDateException e) {
            System.err.println("Booking error: " + e.getMessage());
        }
    }
}
```

Advantages of Using Custom Exceptions in Real-Time Applications

1. **Domain-Specific Errors**: Clearly represent application-specific error conditions (e.g., insufficient stock, invalid age).
2. **Enhanced Debugging**: Provide meaningful messages and context to identify and resolve issues faster.
3. **Improved Readability**: Simplify code by avoiding generic exceptions and making error conditions explicit.
4. **Custom Handling Logic**: Enable targeted handling strategies for specific business scenarios.

These examples illustrate how custom exceptions improve the robustness and maintainability of real-world Java applications.

12.7 SUMMARY

Exception handling is a vital aspect of Java programming, ensuring robust and error-resilient applications. Java provides a rich hierarchy of **built-in exceptions**, such as IOException and NullPointerException, to manage common runtime issues effectively. While these exceptions address generic scenarios, many applications require more specific error representation, making **custom exception subclasses** essential. Developers can create custom exceptions by extending the Exception or RuntimeException classes, providing clear, domain-specific error messages that enhance code readability and debugging. To use exceptions effectively, it is crucial to follow best practices, such as avoiding overuse, providing meaningful error messages, and leveraging checked exceptions for recoverable errors and unchecked exceptions for programming mistakes. By combining built-in and custom exceptions with proper guidelines, developers can design clean, maintainable, and user-friendly error-handling mechanisms tailored to their application's needs.

12.8 TECHNICAL TERMS

Error, exception, try, catch, throw, throws, Checked Exceptio, Unchecked Exception.

12.9 SELF ASSESSMENT QUESTIONS

Essay questions:

1. What are the guidelines for deciding when to use checked and unchecked exceptions?
2. How can custom exceptions improve debugging and code readability? Explain with examples.
3. Illustrate the use of try-catch-finally blocks with a real-world scenario.
4. Discuss the impact of improper exception handling on software quality and performance.
5. Write a program that demonstrates the use of multiple custom exceptions in a single application.
6. Why is it essential to clean up resources in the finally block? Provide an example to support your explanation.

Short Answer Questions:

1. Define built-in exceptions in Java.
2. What is the difference between checked and unchecked exceptions?
3. Give two examples of built-in exceptions in Java.
4. What is the purpose of the Throwable class in Java?
5. What are custom exceptions in Java?

12.10 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

AUTHOR: **Dr. Vasantha Rudramalla**

LESSON- 13

THREAD

OBJECTIVES:

After going through this lesson, you will be able to

- Understand what is 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
- Learn about Thread priorities, daemon thread etc.

STRUCTURE OF THE LESSION:

13.1 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.

13.2 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.

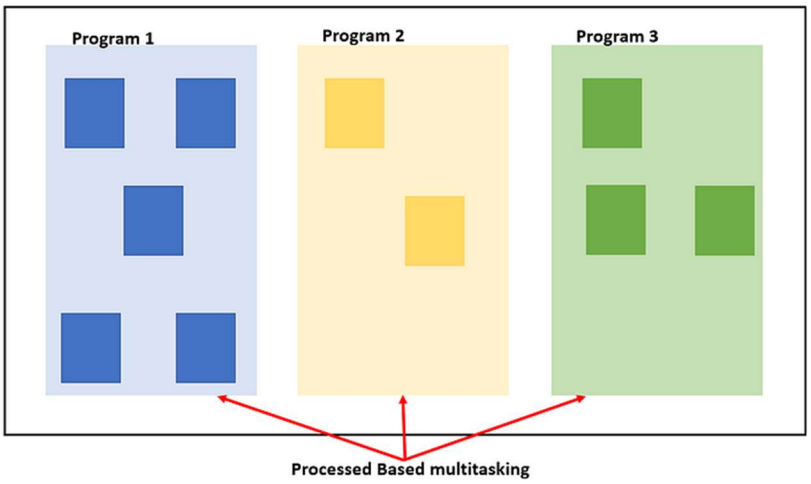


Figure 13.1: Process based multitasking

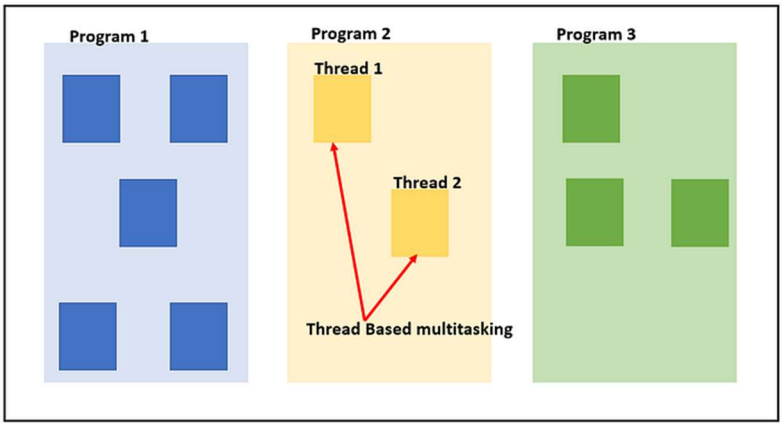


Figure 13.2 Thread based multitasking

Table 13.1 Process based Vs Thread based multitasking

Process-Based Multitasking	Thread – based Multitasking
• 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	• 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

13.3 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.

13.4 THREAD LIFE CYCLE

The Java thread lifecycle refers to the various stages that a thread undergoes from its creation to its termination.

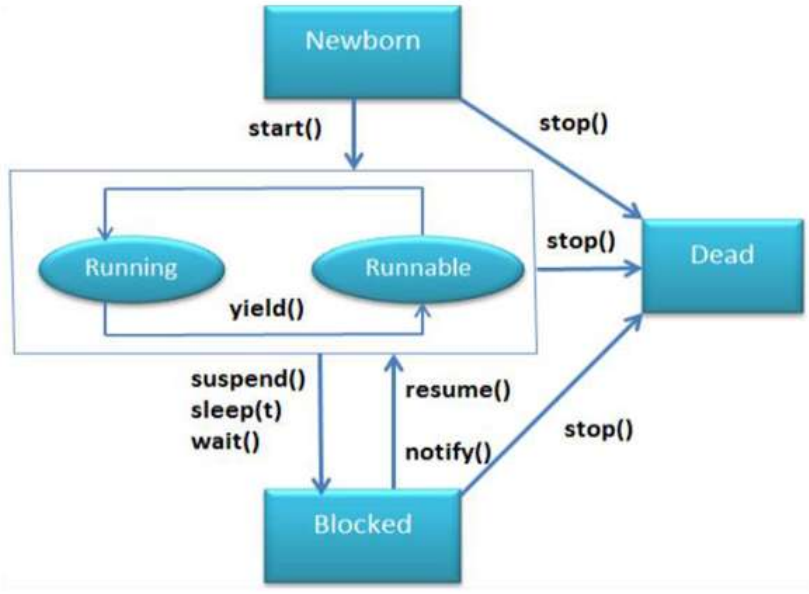


Figure 13.3 Thread life cycle

- **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.

13.5 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.

13.6 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.

❖ **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;
    }
}
```

❖ **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
        }
    }
}
```

13.7 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.

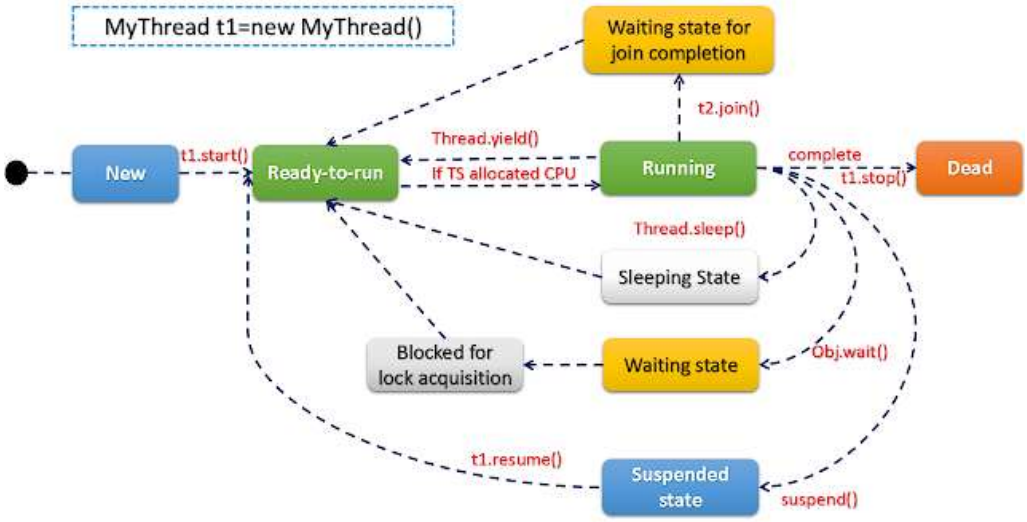


Figure 13.4 : Thread class methods

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()

ontains 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
```

13.8 INTERRUPTING THREADS

Interrupting threads in Java is a mechanism to signal a thread to stop its execution or to perform an alternate action. It is particularly useful when you want to terminate a thread gracefully or handle long-running tasks that might need to be canceled.

Key Methods:

- interrupt(): Signals the thread to interrupt.
- isInterrupted(): Checks if the thread's interrupt flag is set.
- Thread.interrupted(): Checks and clears the interrupt flag.

Example: Interrupting a Thread

```
public class ThreadInterruptionDemo {
    public static void main(String[] args) {
        // Create a thread that performs a long-running task
        Thread worker = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    System.out.println("Working... Step " + i);
                    Thread.sleep(1000); // Simulate a long-running task
                }
            } catch (InterruptedException e) {
                System.out.println("Thread was interrupted! Exiting...");
                return; // Graceful exit
            }
        });

        // Start the thread
        worker.start();

        // Main thread waits for 3 seconds and then interrupts the worker thread
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        worker.interrupt(); // Signal the thread to stop
        System.out.println("Main thread requested interruption.");
    }
}
```

Output

```
Working... Step 1
Working... Step 2
Working... Step 3
```

Main thread requested interruption.

Thread was interrupted! Exiting...

13.9 THREAD PRIORITIES

Thread priorities in Java allow you to control the relative importance of different threads, helping to optimize the performance of a multi-threaded application. Thread priority is an integer value that determines the order in which threads are scheduled by the Java Virtual Machine (JVM). The priority of a thread can influence how much processing time it gets compared to other threads, although the actual behavior depends on the underlying operating system's thread scheduler.

Thread Priority Range

In Java, thread priorities are represented as integer values, with the following range:

- **MIN_PRIORITY**: The lowest priority (value 1).
- **NORM_PRIORITY**: The default priority (value 5).
- **MAX_PRIORITY**: The highest priority (value 10).

These values are constants defined in the Thread class:

- Thread.MIN_PRIORITY = 1
- Thread.NORM_PRIORITY = 5
- Thread.MAX_PRIORITY = 10

You can assign a priority value to a thread by using the setPriority(int priority) method of the Thread class.

You can set the priority of a thread using the setPriority() method:

```
Thread thread = new Thread();
thread.setPriority(Thread.MAX_PRIORITY); // Set to maximum priority
```

```
public class ThreadPriorityExample {

   public static void main(String[] args) {

      // Create two threads with different priorities
      Thread highPriorityThread = new Thread(new Task(), "High Priority Thread");
```

```
        Thread lowPriorityThread = new Thread(new Task(), "Low Priority Thread");

        // Set the priorities of the threads
        highPriorityThread.setPriority(Thread.MAX_PRIORITY); // High priority
        lowPriorityThread.setPriority(Thread.MIN_PRIORITY); // Low priority

        // Start both threads
        highPriorityThread.start();
        lowPriorityThread.start();
    }
    // Task that prints the name of the current thread
    static class Task implements Runnable {
        public void run() {
            // Print the name of the thread
            System.out.println(Thread.currentThread().getName() + " is running");
            try {
                // Sleep for a while to simulate work
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Output:

High Priority Thread is running

Low Priority Thread is running

In this example, you would likely see the "High Priority Thread" run first, as it has a higher priority. However, the exact order is subject to the JVM and the operating system's scheduling behavior.

13.10 SYNCHRONIZING THREADS

Thread synchronization in Java is a critical concept for preventing data inconsistency and ensuring that multiple threads can safely access shared resources. When multiple threads try

to access and modify shared data concurrently, it can lead to problems like race conditions, where the outcome depends on the sequence or timing of the threads' execution. Synchronization helps avoid these issues by controlling access to the shared resource.

In Java, synchronization can be achieved using the **synchronized** keyword. This keyword can be applied in two main ways:

1. **Synchronized Methods**: You can use the synchronized keyword in the method declaration to ensure that only one thread can execute that method at any given time.
2. **Synchronized Blocks**: You can also synchronize specific blocks of code inside a method, allowing more fine-grained control over synchronization.

❖ Synchronized Methods

When a method is declared as synchronized, the thread that invokes the method acquires a lock on the object the method belongs to (or a class lock for static methods). No other thread can execute any synchronized method on the same object until the lock is released.

Example: Synchronized Method

```
class Counter {
   private int count = 0;
   // Synchronized method to ensure thread safety
   public synchronized void increment() {
      count++;
   }

   public int getCount() {
      return count;
   }
}

public class SyncMethodExample {
   public static void main(String[] args) throws InterruptedException {
      Counter counter = new Counter();

      // Create two threads that increment the counter
      Thread t1 = new Thread(() -> {
```

```
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    });

    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    });
    t1.start();
    t2.start();

    t1.join();
    t2.join();
    System.out.println("Final Count: " + counter.getCount());
  }
}
```

In this example, the increment() method is synchronized, ensuring that only one thread can increment the count at a time, preventing data inconsistency. Without synchronization, both threads could attempt to increment count at the same time, leading to incorrect results.

❖ **Synchronized Blocks**

Synchronized blocks provide a more granular approach to synchronization. Instead of synchronizing the entire method, you can synchronize only a specific part of the code. This reduces the performance overhead, especially if the critical section is small.

Example: Synchronized Block

```
class Counter {
  private int count = 0;

  public void increment() {
    // Synchronize only the critical section
    synchronized (this) {
      count++;
```

```
        }
    }

    public int getCount() {
        return count;
    }
}
public class SyncBlockExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        // Create two threads that increment the counter
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("Final Count: " + counter.getCount());
    }
}
```

In this example, the synchronized block is used inside the increment() method. The synchronization happens only on the critical section of the method, where the shared resource (count) is modified. This approach can improve performance when you don't need to synchronize the entire method.

13.11 INTERTHREAD COMMUNICATION

Interthread communication refers to the coordination of multiple threads within a program to work together effectively. In Java, this concept is achieved using special mechanisms that allow threads to communicate and synchronize their activities. When multiple threads need to communicate and share data, interthread communication provides a way to manage the interactions.

In Java, **interthread communication** is built around the concepts of **waiting**, **notifying**, and **notifying all**. The primary mechanism for interthread communication involves the **wait()**, **notify()**, and **notifyAll()** methods, which are defined in the Object class. All objects in Java inherently have the ability to communicate between threads.

Key Concepts of Interthread Communication

1. **Waiting for a Condition**:
    - A thread can enter a waiting state using the wait() method. This causes the thread to release the lock and enter a waiting state until another thread signals it to wake up.
    - The wait() method must be called from within a synchronized block or method.
2. **Notifying Waiting Threads**:
    - The notify() method is used to wake up one of the threads that are waiting on the object's lock.
    - The notifyAll() method wakes up all the threads that are waiting on the lock of the object.
3. **Thread Cooperation**:
    - Interthread communication allows threads to cooperate. For example, one thread may produce data, and another thread may consume it. The producer thread can signal the consumer thread when it has new data, and the consumer thread can wait for data to be available.

Using wait(), notify(), and notifyAll()

These methods must be used within a synchronized block because they depend on the intrinsic lock of the object to work correctly.

- **wait()**: Causes the current thread to wait until another thread sends a notification (via notify() or notifyAll()).
- **notify()**: Wakes up one thread that is waiting on the object.
- **notifyAll()**: Wakes up all threads that are waiting on the object.

Example: Producer-Consumer Problem

This is a classic example of interthread communication, where one thread (the producer) produces data, and another thread (the consumer) consumes that data. The producer and consumer need to synchronize their operations.

```
class SharedResource {
    private int data;
    private boolean available = false;
    // Producer method
    public synchronized void produce(int data) {
        while (available) {
            try {
                wait(); // Wait until the consumer consumes
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        this.data = data;
        available = true;
        System.out.println("Produced: " + data);
        notify(); // Notify consumer that data is available
    }

    // Consumer method
    public synchronized void consume() {
        while (!available) {
            try {
                wait(); // Wait until the producer produces
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Consumed: " + data);
        available = false;
        notify(); // Notify producer that data has been consumed
```

```
    }
}
public class ProducerConsumerExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        // Producer thread
        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.produce(i);
            }
        });

        // Consumer thread
        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.consume();
            }
        });

        producer.start();
        consumer.start();
    }
}

class SharedResource {
    private int data;
    private boolean available = false;

    // Producer method
    public synchronized void produce(int data) {
        while (available) {
            try {
                wait(); // Wait until the consumer consumes
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        this.data = data;
        available = true;
        System.out.println("Produced: " + data);
        notify(); // Notify consumer that data is available
    }

    // Consumer method
    public synchronized void consume() {
        while (!available) {
            try {
                wait(); // Wait until the producer produces
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Consumed: " + data);
        available = false;
        notify(); // Notify producer that data has been consumed
    }
}
public class ProducerConsumerExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        // Producer thread
        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.produce(i);
            }
        });
```

```
        // Consumer thread
        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                resource.consume();
            }
        });

        producer.start();
        consumer.start();
    }
}
```

Interthread communication is a powerful feature in Java that allows threads to cooperate and synchronize their operations. By using wait(), notify(), and notifyAll(), threads can effectively communicate and manage shared resources. This is especially important in scenarios like the **producer-consumer problem** where threads depend on each other's actions to perform their tasks. Proper synchronization ensures that threads interact correctly without causing issues like race conditions, deadlocks, or data inconsistency.

13.12 THREAD GROUPS

In Java, Thread Groups provide a way to organize and manage groups of related threads. A thread group is essentially a container for multiple threads that can be treated as a collective entity. The primary purpose of thread groups is to enable more efficient thread management, such as managing thread behaviors, exception handling, and thread priorities.

Thread groups are represented by the ThreadGroup class in Java, and they are often used in large, complex applications where many threads need to be grouped for easier management.

Key Features of Thread Groups

1. Organization of Threads:
    - Thread groups provide a way to group related threads together. This is useful in applications with multiple components, where each component has its own set of threads.
2. Thread Management:

- By grouping threads, you can set the priority for all threads in the group or handle interruptions and exceptions for the group as a whole.

3. Exception Handling:
   - Thread groups allow you to handle uncaught exceptions for all threads in the group. If any thread in the group throws an uncaught exception, the thread group can take action (e.g., logging, stopping the threads).
4. Controlling Threads:
   - Thread groups allow you to control thread behaviors like setting priorities, listing all threads, checking if threads are alive, and even stopping all threads in a group (though this is not recommended).

Creating and Using Thread Groups

1. Creating a Thread Group

To create a new thread group, you need to instantiate a ThreadGroup object. You can either create a thread group with a specific name or specify a parent thread group.

ThreadGroup group1 = new ThreadGroup("Group 1"); // Creating a thread group with a name

- Parent Thread Group: If no parent is specified, the default thread group is the one associated with the Java virtual machine (JVM). You can create a thread group with a parent by passing the parent group as a parameter:

ThreadGroup parentGroup = new ThreadGroup("Parent Group");

ThreadGroup childGroup = new ThreadGroup(parentGroup, "Child Group");

2. Adding Threads to a Thread Group

Once a thread group is created, you can add threads to it by passing the thread group to the constructor of a Thread:

java

Copy code

```
Thread t1 = new Thread(group1, new Runnable() {
   public void run() {
      System.out.println("Thread in Group 1");
   }
});
t1.start();
```

Alternatively, you can create threads first and then assign them to a thread group later, though this is less common.

3. Managing Threads in a Thread Group

Once you have a thread group, you can manage and control the threads within it using several methods provided by the ThreadGroup class.

- Setting Thread Priorities: You can set a thread priority for all threads in the group using the setMaxPriority(int priority) method.

```
group1.setMaxPriority(Thread.NORM_PRIORITY); // Set max priority for the group
```

- Listing Threads: You can list all the threads in a thread group using the enumerate() method:

```
Thread[] threads = new Thread[group1.activeCount()];
group1.enumerate(threads); // Get the threads in the group
for (Thread t : threads) {
   System.out.println(t.getName());
}
```

- Interrupting Threads: You can interrupt all threads in the group by calling interrupt() on the group:

```
group1.interrupt(); // Interrupts all threads in the group
```

- Checking if a Group is Alive: You can check if any thread in the group is still alive using the activeCount() method, which returns the number of active threads in the group.

```
int count = group1.activeCount(); // Get the active thread count in the group
System.out.println("Active threads: " + count);
```

Example: Using Thread Groups in Java

Here is an example where we create two thread groups and manage the threads within them:

```
public class ThreadGroupExample {
   public static void main(String[] args) {
      // Create a parent thread group
      ThreadGroup group1 = new ThreadGroup("Group 1");

      // Create threads in the thread group
      Thread t1 = new Thread(group1, new Runnable() {
         public void run() {
            System.out.println(Thread.currentThread().getName() + " is running in Group 1");
         }
```

```
        });

        Thread t2 = new Thread(group1, new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName() + " is running in Group 1");
            }
        });
        // Start threads
        t1.start();
        t2.start();

        // List all threads in Group 1
        Thread[] threads = new Thread[group1.activeCount()];
        group1.enumerate(threads);
        for (Thread thread : threads) {
            System.out.println("Thread in Group 1: " + thread.getName());
        }
        // Set maximum priority for threads in the group
        group1.setMaxPriority(Thread.MAX_PRIORITY);

        // Create a child thread group
        ThreadGroup group2 = new ThreadGroup(group1, "Group 2");

        // Create a thread in Group 2
        Thread t3 = new Thread(group2, new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName() + " is running in Group 2");
            }
        });
        t3.start();
        // Interrupt all threads in Group 1
        group1.interrupt();
    }
}
```

Output

Thread-0 is running in Group 1

Thread-1 is running in Group 1

Thread in Group 1: Thread-0

Thread in Group 1: Thread-1

Thread-2 is running in Group 2

In this example, we created two thread groups: Group 1 and Group 2. Threads are assigned to these groups, and the behavior of these threads is managed accordingly. After starting the threads, we listed all the threads in Group 1 and set the maximum priority for the group.

Important Methods of ThreadGroup

Here are some key methods provided by the ThreadGroup class:

- getName(): Returns the name of the thread group.
- activeCount(): Returns the number of active threads in the group.
- activeGroupCount(): Returns the number of active subgroups.
- interrupt(): Interrupts all threads in the group.
- destroy(): Destroys the thread group and all its threads (note: this method is deprecated).
- setMaxPriority(int priority): Sets the maximum priority of threads in the group.

Limitations and Considerations

1. Thread Group Destruction: Thread groups can't be destroyed explicitly in modern Java (using destroy() is deprecated). It’s the JVM's responsibility to clean up thread groups once all threads within them have finished execution.
2. Thread Safety: Like individual threads, thread groups should be used with caution in multi-threaded environments. Since thread groups provide centralized control over threads, there’s potential for conflict if multiple threads are manipulating the group at once.
3. Deprecated Features: Some features related to thread groups, such as managing thread group hierarchies and explicit thread group destruction, are deprecated in favor of using more modern concurrency tools like the java.util.concurrent package.

Thread groups in Java provide a way to organize and manage related threads, offering control over their priorities, interrupting them, and handling exceptions. While thread groups can simplify thread management in certain applications, they are not as commonly used in

modern Java applications, where higher-level concurrency utilities from the java.util.concurrent package are preferred.

13.13 DAEMON THREADS

A daemon thread in Java is a type of thread that runs in the background to perform tasks that are not critical to the application's main functionality. The key characteristic of a daemon thread is that it does not prevent the JVM from exiting when the program has finished executing all non-daemon threads.

Key Features of Daemon Threads:

1. Background Execution:
    - Daemon threads are typically used for background tasks such as garbage collection, monitoring threads, or handling periodic operations like timeouts or cleanup.
    - These threads run continuously as long as the JVM is running, but they do not keep the JVM alive when all non-daemon threads are terminated.
2. JVM Shutdown:
    - The JVM does not wait for daemon threads to complete before it terminates. If only daemon threads are left running, the JVM will terminate the program, which is why daemon threads should be used for tasks that can be interrupted or abandoned.
3. Automatic Termination:
    - When all non-daemon threads finish execution, the JVM shuts down, and any remaining daemon threads are automatically terminated. This makes daemon threads suitable for tasks that can be stopped abruptly without affecting the overall program.

How to Create a Daemon Thread

To create a daemon thread in Java, you must call the setDaemon(true) method on a thread object before it is started. Once the thread is started, its daemon status cannot be changed.

Here is an example demonstrating how to create and use daemon threads:

```
class DaemonThreadExample extends Thread {
  public void run() {
    while (true) {
      try {
        Thread.sleep(1000); // Simulate some work
```

```
            System.out.println("Daemon thread is working...");
        } catch (InterruptedException e) {
            System.out.println("Daemon thread interrupted.");
        }
    }
  }
}
public class Main {
  public static void main(String[] args) {
    // Create a daemon thread
    DaemonThreadExample daemonThread = new DaemonThreadExample();
    daemonThread.setDaemon(true);  // Set as daemon thread

    // Start the daemon thread
    daemonThread.start();

    // Main thread sleeps for a while, allowing daemon thread to run
    try {
      Thread.sleep(5000); // Main thread sleeps for 5 seconds
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Main thread finished execution.");
  }
}
```

1. **Daemon Thread Creation**:
    - A DaemonThreadExample class extends Thread and overrides the run() method to simulate a task that continuously prints a message every second.
    - The thread is marked as a daemon thread by calling setDaemon(true) before starting it.
2. **Main Thread**:
    - The main thread sleeps for 5 seconds, allowing the daemon thread to run in the background.

- After 5 seconds, the main thread completes its execution and the JVM shuts down.
- Since the daemon thread is still running, it is terminated abruptly when the JVM exits.

Daemon Thread Lifecycle

- **Creation**: A thread is created and initially marked as a user thread by default.
- **Setting as Daemon**: By calling setDaemon(true), a thread can be designated as a daemon thread. This must be done before the thread is started.
- **Running**: The daemon thread runs just like a normal thread, performing background tasks.
- **Termination**: When all non-daemon threads have finished execution, the JVM will terminate the program, and daemon threads will be forcefully stopped.

Important Points About Daemon Threads:

1. **Set Daemon Before Start**: The setDaemon(true) method must be called before the thread is started. Attempting to change the daemon status after the thread is started will throw an IllegalThreadStateException.
2. **Daemon Thread Terminates Automatically**: The JVM does not wait for daemon threads to finish before exiting. When all non-daemon threads are finished, the JVM terminates the program, and daemon threads are stopped immediately, even if they haven't finished their execution.
3. **Use with Caution**: Since daemon threads are terminated abruptly when the JVM shuts down, they are typically used for background tasks like garbage collection, housekeeping tasks, or monitoring services. They should not be used for tasks that require completion before the program terminates, as they may leave resources uncleaned or processes incomplete.
4. **Non-daemon Threads**: In contrast to daemon threads, **non-daemon threads** (also known as user threads) must complete their execution before the JVM can terminate. These threads are used for critical tasks that need to finish execution.

Daemon Thread Example: Garbage Collection

The JVM uses daemon threads for automatic garbage collection. The garbage collector is a background task that runs in a daemon thread, reclaiming memory from objects that are no longer reachable. If there are no non-daemon threads left, the JVM will shut down the application, terminating any remaining daemon threads, including the garbage collector.

Daemon threads are a powerful feature in Java that help manage background tasks without affecting the program's main functionality. They run in the background and do not prevent the JVM from shutting down when all non-daemon threads have completed their execution. While daemon threads are useful for tasks like periodic monitoring, cleanup, and garbage collection, they should be used carefully, as they can be terminated abruptly when the JVM exits.

13.14 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.

13.15 TECHNICAL TERMS

Deamon thread, Priority, Garbage Collection, Life Cycle

13.16 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?

13.17 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

AUTHOR: **Mrs. Appikatla Pushpa Latha**

LESSON- 14

EVENT HANDLING

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the Concept of Events
- Identify Event Sources
- Learn About Event Listeners
- Understand the Relationship Between Event Sources and Listeners
- Handle Mouse and Keyboard Events
- Utilize Adapter Classes for Simplified Event Handling

STRUCTURE OF THE LESSION:

14.1 INTRODUCTION

In Java, event handling is a fundamental concept that enables user interaction with graphical user interfaces (GUIs) and other event-driven applications. At the core of event handling are events, which represent actions or occurrences, such as a button click or mouse movement. These events are generated by event sources, which are components that detect user actions, like buttons, text fields, or mouse movements. When an event occurs, it is passed to event listeners, which are interfaces that define methods to handle specific types of events. The relationship between event sources and listeners is one of communication, where an event source sends events to a listener, which then responds by executing appropriate actions. Java utilizes the delegation event model, where event handling is decoupled from the components generating the events, allowing more flexibility and maintainability. Events can be categorized as semantic events, which represent high-level user actions (like a button press), and low-level events, which represent detailed actions such as mouse movements or key presses. For instance, handling a button click involves detecting the click event and triggering the appropriate action, while handling mouse and keyboard events requires more specialized listeners like MouseListener and KeyListener. Adapter classes, such as MouseAdapter or KeyAdapter, provide default implementations of listener methods, making event handling more convenient by allowing developers to override only the methods they need. This event-driven programming model ensures that applications can respond dynamically to user interactions.

14.2 EVENT

In Java, events represent the occurrences or actions that happen during the execution of a program, often triggered by user interactions with the graphical user interface (GUI) components. These events can include actions like button clicks, mouse movements, keyboard presses, and other activities that require the program to respond.

An event in Java is a signal that something has happened, such as:

- A user clicking a button.
- The user pressing a key on the keyboard.
- A mouse movement or button click.

Types of Events

Events can be categorized into **semantic** and **low-level events**:

1. **Semantic Events**:
    - These are high-level events that represent user actions, such as pressing a button, selecting a menu item, or closing a window.

- **Example**: A ButtonClickEvent when a user clicks a button on the interface.

2. **Low-level Events**:
    - These events represent more granular, detailed actions, such as mouse movements, key presses, and mouse clicks.
    - **Example**: A MouseEvent when the user moves the mouse or clicks on a component.

14.3 EVENT SOURCES

In Java, an **event source** is an object or component that generates events when a user interacts with it. These components could be graphical user interface (GUI) elements, such as buttons, text fields, checkboxes, or even non-GUI elements like timers. The event source is responsible for detecting user actions and triggering corresponding events that are handled by event listeners.

Role of Event Sources

- **Detect User Interaction**: Event sources are responsible for detecting user actions, such as clicks, key presses, or mouse movements.
- **Generate Events**: When an interaction occurs, the event source creates an event object that describes the action (e.g., which button was clicked or which key was pressed).
- **Notify Event Listeners**: The event source sends the generated event to the appropriate event listener, which then responds to the event by executing specific code.

Common Event Sources in Java

Java provides a variety of GUI components that can serve as event sources, and these components are usually part of the javax.swing or java.awt packages. Some common event sources include:

1. **Buttons** (JButton, Button):
    - A button is an event source that generates an event when clicked by the user.

- **Example**: When the user clicks a button, an ActionEvent is triggered.

2. **Text Fields** (JTextField, TextField):
   - A text field can generate events such as when the user enters text or presses the Enter key.
   - **Example**: A KeyEvent is generated when the user presses a key inside a text field.
3. **Check Boxes** (JCheckBox, Checkbox):
   - Checkboxes are event sources that trigger events when the user selects or deselects them.
   - **Example**: An ItemEvent is generated when the state of a checkbox changes.
4. **Menus** (JMenu, Menu):
   - Menus can generate events when a menu item is selected.
   - **Example**: A MenuEvent or ActionEvent is triggered when the user selects a menu item.
5. **Mouse Events** (JComponent, Component):
   - Various components like panels or buttons can serve as event sources for mouse-related events such as mouse clicks, movements, or drags.
   - **Example**: A MouseEvent is generated when the user clicks the mouse on a component.
6. **Window Events** (Window, JFrame):
   - Components like windows or frames can generate events when a window is opened, closed, resized, or activated.
   - **Example**: A WindowEvent is generated when a window is closed.
7. **Timers** (Timer):
   - Timers in Java generate action events after a specified time interval.

- **Example**: A TimerEvent is triggered when the timer reaches the specified delay.

How Event Sources Work

The work of an event source can be summarized in the following steps:

1. **User Interaction**: A user interacts with a GUI component. For instance, a user clicks a button or types in a text field.
2. **Event Generation**: The event source (e.g., a button or text field) generates an event that encapsulates information about the user action (such as the mouse location, the key pressed, or the button clicked).
3. **Event Listener Registration**: The event source must have event listeners registered to handle the events it generates. This is done using methods like addActionListener(), addMouseListener(), or addKeyListener().
4. **Event Handling**: The event listener method is invoked to handle the event, typically executing a specific action (e.g., updating the UI, performing a calculation, etc.).

14.4 EVENT CLASSES

In Java, event classes are used to encapsulate details about the events that occur in an application. These classes store the information about the user action or system event and are passed from the event source (such as a button, text field, or mouse) to the event listener for handling. Event classes are part of the java.awt.event or javax.swing.event package, and they represent different types of actions that occur during the execution of a program.

Each event class typically extends to the java.util.EventObject class, and they contain information relevant to the specific type of event, such as which component generated the event, the type of event, or additional details like the mouse position or the key pressed.

Common Event Classes in Java

Java provides several event classes that correspond to different types of events. Some commonly used event classes include:

1. **ActionEvent**:

- Represents a high-level action event, such as clicking a button or selecting a menu item.

2. **MouseEvent**:
   - Represents events generated by mouse actions such as clicking, moving, or pressing the mouse button.
3. **KeyEvent**:
   - Represents events triggered by keyboard actions, such as pressing or releasing a key.
4. **WindowEvent**:
   - Represents events triggered by actions on a window, such as opening, closing, or resizing a window.
5. **FocusEvent**:
   - Represents events related to component focus, such as when a component gains or loses focus.

Event classes in Java are integral to the event-handling mechanism, encapsulating the details of various user actions. They are designed to be passed from the **event source** to the **event listener**, where appropriate actions can be performed. By understanding and utilizing the appropriate event classes, Java developers can build responsive applications that react to user interactions efficiently.

14.5 EVENT LISTENERS

In Java, event listeners have interfaces that define methods to handle specific types of events. They allow a program to respond to user actions or other events generated by components such as buttons, text fields, checkboxes, and other GUI elements. Event listeners are a crucial part of the event-driven programming model, where the program waits for user interactions (such as button clicks or keyboard presses) and handles them accordingly.

Event listeners are designed to respond to particular events by providing methods that are called when those events occur. For example, an

ActionListener is triggered when a user clicks a button or presses a menu item, while a MouseListener is invoked when the user interacts with the mouse.

Common Event Listener Interfaces in Java

Here are some commonly used event listener interfaces in Java:

1. **ActionListener**:
    - **Purpose**: Listens for action events, such as when a user clicks a button or selects a menu item.
    - **Key Method**: actionPerformed(ActionEvent e)
2. **MouseListener**:

- **Purpose**: Listens for mouse-related events such as mouse clicks, entering, exiting, and pressing mouse buttons.
- **Key Methods**: mouseClicked(MouseEvent e), mousePressed(MouseEvent e), mouseReleased(MouseEvent e), mouseEntered(MouseEvent e), mouseExited(MouseEvent e)

KeyListener:

- **Purpose**: Listens for keyboard events such as key presses and releases.
- **Key Methods**: keyPressed(KeyEvent e), keyReleased(KeyEvent e), keyTyped(KeyEvent e)

WindowListener:

- **Purpose**: Listens for window events such as opening, closing, activating, or deactivating a window.
- **Key Methods**: windowOpened(WindowEvent e), windowClosing(WindowEvent e), windowClosed(WindowEvent e), windowIconified(WindowEvent e), windowDeiconified(WindowEvent e), windowActivated(WindowEvent e), windowDeactivated(WindowEvent e)

ItemListener:

- **Purpose**: Listens for changes in item selection, such as when a checkbox or radio button is selected or deselected.

- **Key Method**: itemStateChanged(ItemEvent e)

14.6 RELATIONSHIP BETWEEN EVENT SOURCES AND LISTENERS

The relationship between event sources and listeners is fundamental to the event-handling mechanism in Java. The **event source** generates events, while the **event listener** responds to those events. The source and listener are loosely coupled, allowing flexibility in how events are handled. This decoupling also enables reusability, as the same listener can be used with different event sources. Properly managing this relationship is key to building interactive and responsive GUI applications in Java.

How the Relationship Works

1. **Event Source Generates an Event**:
    - When a user interacts with a component (e.g., clicking a button or typing in a text field), the event source generates an event object. For example, if a user clicks a button, the button generates an ActionEvent.
2. **Event Listener Registers with the Event Source**:
    - To handle the event, the event listener must be registered with the event source. This is done by calling the addListener() method (such as addActionListener(), addMouseListener(), etc.) of the event source and passing the listener object that will handle the event.
3. **Event Listener Handles the Event**:
    - Once registered, the listener is notified when the event occurs. The listener's method (such as actionPerformed() for an ActionListener, mouseClicked() for a MouseListener, or keyPressed() for a KeyListener) is invoked, and the program responds to the event, typically by performing some action or updating the user interface.
4. **Decoupling of Event Source and Listener**:
    - The relationship between the event source and the event listener is decoupled. The event source does not need to know anything about what the listener does when it handles the event. It simply notifies the listener that the event has occurred. Similarly, the listener does not need to know the specific details of the event source; it only handles the events it is interested in.

14.7 DELEGATION EVENT MODEL

The **Delegation Event Model** is the primary event-handling mechanism used in Java's Abstract Window Toolkit (AWT) and Swing for managing events in graphical user interface (GUI) applications. It is based on the principle of event delegation, where the responsibility for handling events is "delegated" to event listener objects. The model is designed to simplify the process of handling user input and interactions in Java applications.

Advantages of the Delegation Event Model

1. **Loose Coupling**:
   - The Delegation Event Model separates the concerns of the event source and the event listener. The event source only generates events and does not need to know how the event is handled. The event listener responds to the event without modifying the event source, promoting loose coupling between components.
2. **Code Reusability**:
   - The same event listener can be reused for different event sources. For example, a single ActionListener can handle events from multiple buttons, and the same MouseListener can be attached to various components.
3. **Simplified Event Handling**:
   - The model simplifies event handling by allowing the developer to focus on writing event-handling code in the listener methods rather than embedding complex event-handling logic within the event sources themselves.
4. **Event Propagation**:
   - The model also supports event propagation, where events can be passed to other listeners or components. This makes it possible to create complex event-handling mechanisms, such as event filtering or handling.

Example of the Delegation Event Model

```
import javax.swing.*;
import java.awt.event.*;

public class DelegationEventModelExample {
   public static void main(String[] args) {
      // Create a JFrame
      JFrame frame = new JFrame("Delegation Event Model Example");
```

```
        // Create a JButton (Event Source)
        JButton button = new JButton("Click Me");

        // Add an ActionListener to the button (Event Listener)
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // Event Handler: Handling the event (Button Click)
                System.out.println("Button was clicked!");
            }
        });
        // Set up the frame
        frame.add(button);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

14.8 SEMANTIC AND LOW-LEVEL EVENTS

In Java, particularly in GUI programming, events can be categorized into **semantic events** and **low-level events** based on the type of information they carry and the context in which they are used. Understanding these two categories helps developers handle different types of interactions and manage event-driven programming efficiently.

Semantic events are high-level events that are directly related to user actions or tasks that the user intends to perform. These events are typically associated with common actions in a graphical user interface (GUI), such as clicking a button, selecting a menu item, or submitting a form. They represent logical or meaningful actions that the user wants to perform and are part of the overall flow of interaction in an application.

Examples of Semantic Events:

- **ActionEvent**: Triggered when a user clicks a button or selects a menu item. The ActionListener interface handles this event.
- **ItemEvent**: Triggered when the state of a checkbox or menu item changes, such as being selected or deselected. The ItemListener interface handles this event.

- **WindowEvent**: Triggered when a window is opened, closed, or resized. The WindowListener interface handles these events.

Low-level Events

Low-level events are more detailed and granular events that represent user interactions at a much closer level, often capturing basic hardware-level operations or more specific actions that occur on a component. These events focus on the raw inputs that generate actions, such as mouse movements, key presses, and mouse button clicks. Low-level events provide more detailed information about the user's interaction with the system, which can be useful for tasks that require finer control or monitoring.

Examples of Low-level Events:

- **MouseEvent**: Captures events related to mouse actions, such as mouse clicks, mouse movements, and mouse dragging. Handled by the MouseListener and MouseMotionListener interfaces.
- **KeyEvent**: Captures events related to keyboard actions, such as key presses and key releases. Handled by the KeyListener interface.
- **FocusEvent**: Captures events related to changes in the focus of a component, such as gaining or losing focus. Handled by the FocusListener interface.

Table 14.1 Comparison Between Semantic and Low-level Events

Aspect	Semantic Events	Low-level Events
Level of Abstraction	High-level events representing user actions.	Low-level events capturing detailed input.
Event Type	Typically related to user actions (e.g., button click, menu selection).	Related to basic input (e.g., mouse movements, key presses).
Examples	ActionEvent, ItemEvent, WindowEvent.	MouseEvent, KeyEvent, FocusEvent.
Use Case	For handling high-level tasks like executing commands or navigating.	For capturing fine-grained user interactions, such as tracking mouse movement or detecting key presses.
Event Listener	Common listeners like ActionListener, ItemListener.	Low-level listeners like MouseListener, KeyListener, FocusListener.

14.9 EXAMPLES HANDLING A BUTTON CLICK, HANDLING MOUSE AND KEYBOARD EVENTS

In Java, events such as button clicks, mouse actions, and keyboard inputs are typically handled using listeners. Below are examples demonstrating how to handle each of these events in a simple Java application.

❖ Handling a Button Click (ActionEvent)

A **button click** is one of the most common user interactions in a GUI application. The event generated by clicking a button is an **ActionEvent**.

Example Code (Button Click - ActionEvent):

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonClickExample {
    public static void main(String[] args) {
        // Create a JFrame
        JFrame frame = new JFrame("Button Click Example");

        // Create a JButton
        JButton button = new JButton("Click Me!");

        // Add ActionListener to handle button click
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Handling the button click event
                JOptionPane.showMessageDialog(frame, "Button was clicked!");
            }
        });

        // Set up the frame layout
        frame.add(button);
        frame.setSize(300, 200);
```

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

- The ActionListener is added to the JButton (button).
- When the button is clicked, the actionPerformed() method is called.
- A dialog is shown using JOptionPane.showMessageDialog() to notify the user that the button was clicked.

❖ Handling Mouse Events (MouseEvent)

Mouse events in Java, such as clicks or movement, are handled by implementing a MouseListener or MouseMotionListener. Below is an example of handling a mouse click event using MouseListener.

Example Code (Mouse Click - MouseEvent):

```
import javax.swing.*;
import java.awt.event.*;
public class MouseEventExample {
    public static void main(String[] args) {
        // Create a JFrame
        JFrame frame = new JFrame("Mouse Event Example");

        // Create a JLabel
        JLabel label = new JLabel("Click anywhere on the screen");

        // Add MouseListener to handle mouse clicks
        label.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                // Handling mouse click event
                System.out.println("Mouse clicked at coordinates: " + e.getX() + ", " + e.getY());
            }
        });
        // Set up the frame layout
        frame.add(label);
```

```
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

- A MouseListener is added to the JLabel (label).
- When the user clicks anywhere on the label, the mouseClicked() method is invoked, and it prints the mouse click coordinates to the console.

❖ Handling Keyboard Events (KeyEvent)

Keyboard events are captured using the KeyListener interface, which detects when keys are pressed or released. Here's how to handle keyboard events.

Example Code (Key Press - KeyEvent):

```
import javax.swing.*;
import java.awt.event.*;
public class KeyEventExample {
    public static void main(String[] args) {
        // Create a JFrame
        JFrame frame = new JFrame("Keyboard Event Example");
        // Create a JTextField to capture keyboard input
        JTextField textField = new JTextField(20);
        // Add KeyListener to handle key presses
        textField.addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                // Handling the key press event
                System.out.println("Key pressed: " + e.getKeyChar());
            }
            @Override
            public void keyReleased(KeyEvent e) {
                // Handling key release event
                System.out.println("Key released: " + e.getKeyChar());
            }
        });
```

```
        // Set up the frame layout
        frame.add(textField);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

- The KeyListener is added to a JTextField (textField).
- The keyPressed() method is triggered when a key is pressed, and the keyReleased() method is called when the key is released.
- The key character is printed to the console to indicate which key was pressed or released.

14.10 ADAPTER CLASSES

n Java, **adapter classes** are used to simplify event handling by providing default implementations for the methods of an event listener interface. These classes are particularly useful when you only need to implement a subset of the methods in an interface, avoiding the need to implement all the methods, many of which may not be needed in a specific scenario.

Java provides several **adapter classes** that implement the listener interfaces with empty method bodies, allowing you to override only the methods that you are interested in. These adapter classes are part of the **Java AWT** and **Swing** libraries.

Purpose of Adapter Classes

- **Simplification**: They simplify the code by eliminating the need to provide empty method implementations for the unused methods of an interface.
- **Code Readability**: By using an adapter, you only need to implement the methods you care about, improving the readability and maintainability of the code.

- **Reduced Boilerplate**: Without adapters, you would have to implement all methods in the listener interfaces, even if some of them are not needed.

Common Adapter Classes in Java

Here are some of the commonly used adapter classes in Java:

- **MouseAdapter**: Simplifies mouse event handling (e.g., MouseListener, MouseMotionListener).
- **KeyAdapter**: Simplifies keyboard event handling (e.g., KeyListener).
- **WindowAdapter**: Simplifies window event handling (e.g., WindowListener).
- **FocusAdapter**: Simplifies focus event handling (e.g., FocusListener).
- **ComponentAdapter**: Simplifies component event handling (e.g., ComponentListener).

Example of Using an Adapter Class

MouseAdapter Example

In the case of mouse events, the MouseAdapter class can be used to simplify handling different mouse events, like clicks and movements.

Here is an example of using MouseAdapter to handle mouse clicks:

```
import javax.swing.*;
import java.awt.event.*;
public class MouseAdapterExample {
    public static void main(String[] args) {
        // Create a JFrame
        JFrame frame = new JFrame("Mouse Adapter Example");
        // Create a JLabel
        JLabel label = new JLabel("Click anywhere on the frame");
        // Use MouseAdapter to handle mouse events
        label.addMouseListener(new MouseAdapter() {
            @Override
```

```
        public void mouseClicked(MouseEvent e) {
            // Handling mouse click event
            System.out.println("Mouse clicked at position: " + e.getX() + ", " +
e.getY());
        }
    });
    // Set up the frame layout
    frame.add(label);
    frame.setSize(300, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

- The MouseAdapter class is used to override only the mouseClicked() method, which is triggered when the user clicks on the JLabel.
- The other methods in the MouseListener interface (like mousePressed(), mouseReleased(), mouseEntered(), and mouseExited()) are not required for this example, so we don't need to implement them.

Example of Using KeyAdapter

Similarly, the KeyAdapter class can be used for keyboard event handling when you don't need to implement all the methods of the KeyListener interface.

```
import javax.swing.*;
import java.awt.event.*;
public class KeyAdapterExample {
  public static void main(String[] args) {
    // Create a JFrame
    JFrame frame = new JFrame("Key Adapter Example");
    // Create a JTextField
    JTextField textField = new JTextField(20);
```

```
        // Use KeyAdapter to handle key events
        textField.addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                // Handling key press event
                System.out.println("Key pressed: " + e.getKeyChar());
            }
        });
        // Set up the frame layout
        frame.add(textField);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

- The KeyAdapter class is used to handle the keyPressed() method, which is triggered when the user presses a key in the JTextField.
- The keyReleased() and keyTyped() methods are not implemented, as they are not needed for this specific functionality.

Advantages of Using Adapter Classes

1. **Reduced Boilerplate Code**: Adapter classes eliminate the need for implementing every method in an interface, making the code shorter and more manageable.
2. **Simplified Event Handling**: When dealing with multiple events from the same listener type (e.g., mouse events), adapter classes allow you to implement only the specific methods you need without being forced to define unused methods.
3. **Improved Readability**: By focusing only on the relevant event handling methods, adapter classes make the code more readable and less cluttered.

4. **Maintainability**: With fewer lines of code and only relevant methods being implemented, maintenance becomes easier.

Adapter classes in Java are a powerful feature that simplifies the process of handling events in GUI applications. By using adapter classes like MouseAdapter, KeyAdapter, and others, you can reduce the amount of boilerplate code required for implementing event listeners, allowing you to focus only on the specific event handling methods that are relevant to your application.

14 .11 SUMMARY

Event handling in Java is a fundamental concept used to respond to user interactions with graphical user interfaces (GUIs). It involves several key components, such as **events**, which are the actions triggered by user input, and **event sources**, which are the components (like buttons, mouse, or keyboard) that generate these events. **Event classes** define the details of the events, while **event listeners** are used to detect and respond to these events. The **relationship between event sources and listeners** is built on the delegation event model, where sources generate events, and listeners handle them. Events can be classified into **semantic** and **low-level** events, with semantic events representing high-level user actions (like button clicks) and low-level events capturing more detailed user input (like mouse movements or key presses). **Adapter classes** are used to simplify event handling by providing default implementations for listener interfaces, allowing developers to implement only the methods they need. Examples of event handling include handling **button clicks** through action events, **mouse events** with MouseListener, and **keyboard events** with KeyListener. This structure allows Java applications to effectively manage user interactions in a modular and organized way.

14.12 TECHNICAL TERMS

Event , Event Sources, Listener, Adapter, Mouse Event, Action Event ,Key Event

14.13 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain the concept of events, event sources, and event classes in Java with examples, and describe how they are used in event handling.
2. Describe the relationship between event sources and listeners in Java. How does the Delegation Event Model facilitate this relationship?

3. Compare and contrast semantic and low-level events in Java. Provide examples of each type and explain their significance in event handling.
4. Provide an example of handling a button click event in Java using ActionListener, and explain the flow of event handling.
5. Discuss how mouse events and keyboard events are handled in Java. Provide code examples for each and explain the differences in handling them.

Short Answer Questions:

1. What is the difference between semantic and low-level events in Java?
2. How does the Delegation Event Model work in Java?
3. What is the role of an event listener in Java event handling?
4. How are mouse events handled in Java? Provide an example.
5. What is the purpose of adapter classes in Java event handling?

14.14 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

AUTHOR: **Mrs. Appikatla Pushpa Latha**

LESSON- 15

APPLETS

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the Inheritance Hierarchy for Applets:
- Differentiate Between Applets and Applications:
- Understand the Life Cycle of an Applet:
- Learn to Develop and Test Applets:

STRUCTURE OF THE LESSION:

15.1 INTRODUCTION

Applets are small Java programs designed to be embedded in web pages and run in a web browser. They extend from the Applet class, which is part of the java.applet package, and inherit various methods to handle their behavior. The inheritance hierarchy for applets involves the Applet class as the base, from which other applet classes are derived. Unlike standalone Java applications, applets are executed in a browser or applet viewer, and they rely on specific lifecycle methods to manage their execution. These methods include init(), which initializes the applet, start(), which starts the applet's execution, stop(), which pauses it, and destroy(), which cleans up resources before the applet is removed. Developing and testing applets requires using an applet viewer or browser to ensure correct behavior and integration into the web page. Additionally, applets can receive parameters passed from HTML pages, which are accessed using the getParameter() method. However, due to security concerns, applets are subject to strict security restrictions, preventing access to sensitive system resources. These security issues are managed through Java's sandbox model, which limits applet actions to protect the user's machine from potential malicious behavior.

15.2 APPLET

An applet in Java is a small application designed to be embedded within a web page and run in a web browser. Applets are primarily used to provide interactive features on websites, such as animations, games, or user interfaces. Unlike standalone Java applications, which are executed directly from the command line, applets run inside a web browser or applet viewer, and they rely on specific methods and life cycle management for execution.

Key Features of Applets:

1. **Execution Environment:**
   - Applets are executed within a Java-enabled web browser or an applet viewer. They run as part of a webpage, which means they can interact with HTML content and be used for web-based applications.
2. **No Main Method:**
   - Unlike standalone Java applications, applets do not have a main() method. Instead, applets use a set of predefined methods that control their life cycle, such as init(), start(), stop(), and destroy().

3. Security Model:

- Applets are executed in a restricted environment (known as a sandbox) to prevent them from accessing sensitive system resources. This ensures the applet does not perform malicious activities such as file manipulation or network access.

15. 3 INHERITANCE HIERARCHY FOR APPLETS

In Java, applets are part of the Java java.applet package and follow a well-defined inheritance hierarchy. The inheritance structure for applets is designed to provide basic functionality and behavior that can be extended to create custom applets.

Here's a breakdown of the key classes in the applet inheritance hierarchy:

1. Object Class (Root Class)

- Location: java.lang.Object
- The root class of every Java class, including applets, is Object. All Java classes inherit from Object, so any applet class will also inherit methods like equals(), hashCode(), toString(), etc., from this class.

2. Applet Class (Base Class for Applets)

- Location: java.applet.Applet
- The core class for creating applets is the Applet class, which extends the Panel class (from the AWT library). This class provides the basic framework for applet functionality. It includes methods such as init(), start(), stop(), and destroy(), which manage the life cycle of the applet.
    - Key Methods:
        - init(): Called when the applet is first loaded.
        - start(): Called when the applet is started or resumed.
        - **stop(): Called when the applet is stopped or suspended.**
        - destroy(): Called when the applet is destroyed.

3. Panel Class (AWT Component)

- Location: java.awt.Panel

- The Applet class extends Panel from the Abstract Window Toolkit (AWT). The Panel class is a container used to group components like buttons, labels, and text fields. Since applets are designed to run inside a window, Panel provides the visual layout and components needed for applet-based applications.

4. Container Class (AWT Container)

- Location: java.awt.Container
- The Panel class, as a container, extends from Container. The Container class allows the arrangement of multiple components within the same window, enabling the applet to manage and display GUI elements.

5. Component Class (AWT Base for All Visual Elements)

- Location: java.awt.Component
- Component is the base class for all AWT components that handle user interaction or display. All visual components, such as buttons, text fields, and labels, are derived from Component, which Container (and consequently Panel) also extends.

6. JApplet Class (Swing Applet)

- Location: javax.swing.JApplet
- The JApplet class is part of the Swing library and is an extension of the Applet class, specifically for creating applets with Swing components (like JButton, JLabel, etc.). It is used when you want to create more advanced user interfaces with Swing rather than AWT components.
    - Difference from Applet: While Applet uses AWT components, JApplet allows for the use of Swing components for more modern UI elements.
- Applet is the base class for traditional applets, providing the fundamental methods for initialization and execution.
- JApplet provides enhanced capabilities through Swing, which are more flexible and capable of creating modern user interfaces.

- Both classes provide a starting point for creating Java applets, with JApplet offering more modern UI capabilities compared to the older Applet class.

This hierarchy allows developers to extend the functionality of applets by creating subclasses that inherit the applet life cycle methods and customize them to handle specific user interactions or interface designs.

15.4 DIFFERENCES BETWEEN APPLETS AND APPLICATIONS

Table 15.1 summarizing the differences between applets and applications

Feature	Applet	Application
Execution Environment	Runs in a web browser or applet viewer	Runs independently on a Java Virtual Machine (JVM)
Entry Point	No main() method; uses life cycle methods (init(), start(), stop(), destroy())	Has a main() method as the entry point for execution
User Interface	Embedded within a web page, using AWT or Swing components	Can have full control over the UI, standalone windows
Life Cycle	Managed by the browser or applet viewer	Controlled by the main() method and program flow
Security	Runs in a restricted "sandbox" environment to limit access to system resources	Has fewer security restrictions, can access system resources with appropriate permissions
Deployment	Deployed in HTML pages using <applet> tag, requires a Java-enabled browser	Distributed as standalone applications (e.g., JAR files, executable files)
Usage	Primarily used for	Used for standalone

	interactive content in web pages (e.g., games, forms)	applications, including desktop software, server-side apps, etc.
Browser Dependency	Requires a Java-enabled browser with an applet viewer or plugin	No browser required, can be run directly from the command line or IDE
Performance	Limited performance due to browser constraints and security sandbox	Can utilize full system resources and hardware, leading to better performance
Interaction with System	Limited system access due to security restrictions	Can interact with the local file system, network, and hardware (with permissions)

15.5 LIFE CYCLE OF AN APPLET

The life cycle of an applet in Java is defined by a set of methods that the applet container (usually a web browser or an applet viewer) calls during its life span. These methods control the applet's behavior from the time it is initialized until it is destroyed. An applet does not have a main() method like a typical Java application. Instead, it follows a specific life cycle that allows it to interact with the user, display content, and handle events in a controlled environment.

The key methods involved in the applet life cycle are:

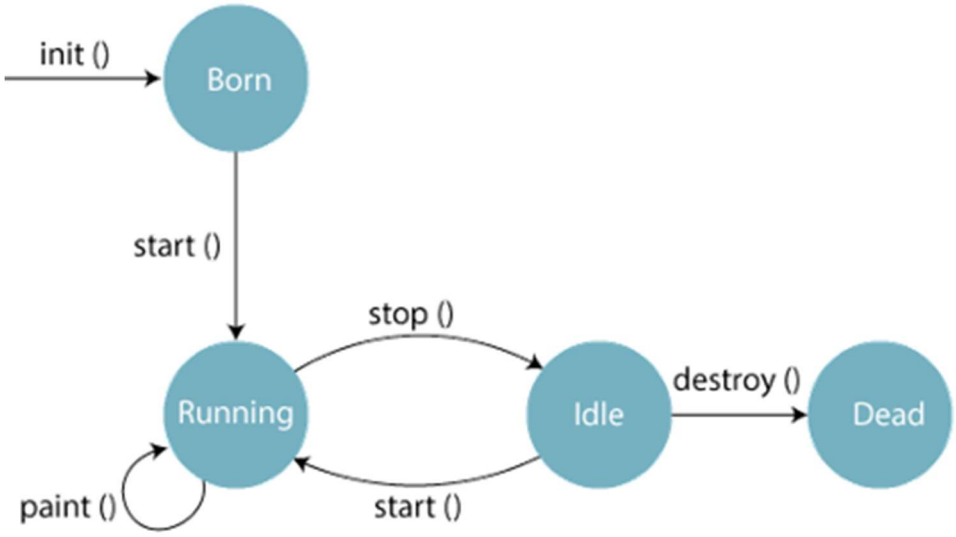


Fig 15.1 Life Cycle of Applet

1. init() Method:

- **Purpose**: This is the first method that is called when an applet is loaded. It is used to initialize the applet's resources, such as setting up the user interface, establishing network connections, or loading configuration files.
- **When is it called?**: The init() method is called once, when the applet is first loaded into memory by the applet container (i.e., the browser or applet viewer).
- **Common Usage**:
  - Set up the applet's user interface.
  - Initialize any resources the applet needs.

```
public void init() {
   // Initialization code here
   System.out.println("Applet initialized");
}
```

2. start() Method:

- **Purpose**: The start() method is called after the init() method. It is called whenever the applet is started or resumed after being paused (for example, when the user returns to a page containing the applet). This method is used for tasks like starting animations or any continuous behavior that the applet should begin when it becomes active.
- **When is it called?**: After init() is called, and every time the applet's page is revisited or the browser refreshes.
- **Common Usage**:
  - Start animations or threads.
  - Begin tasks that should run while the applet is active.

```
java
Copy code
public void start() {
   // Start any animation or thread
   System.out.println("Applet started");
}
```

3. stop() Method:

- **Purpose**: The stop() method is called when the applet is no longer active, for example, when the user navigates away from the page containing the applet, or the applet is explicitly stopped. This method is used to stop activities such as animations, threads, or other ongoing processes that should be halted when the applet is not active.

- **When is it called?**: When the applet is paused or the user leaves the page.
- **Common Usage**:
  - Stop animations or threads.
  - Release resources or halt activities that should not continue when the applet is inactive.

```
public void stop() {
   // Stop any ongoing tasks like animation
   System.out.println("Applet stopped");
}
```

4. destroy() Method:

- **Purpose**: The destroy() method is called when the applet is about to be unloaded from memory, usually when the user navigates away from the applet's page or the applet viewer closes. This method is used to clean up any resources that the applet might have allocated, such as closing files or terminating network connections.
- **When is it called?**: Just before the applet is destroyed or removed from memory.
- **Common Usage**:
  - Close network connections.
  - Release resources such as files or database connections.
  - Clean up any remaining threads or tasks.

```
public void destroy() {
   // Clean up resources here
   System.out.println("Applet destroyed");
}
```

Table Summary of Applet Life Cycle Methods

Method	Purpose	When Called
init()	Initializes the applet (sets up UI, resources)	Once when the applet is loaded
start()	Starts or resumes operations like animations, threads, or interactions	After init(), every time applet becomes visible
stop()	Pauses operations (stops animations, threads)	When applet is no longer visible or becomes inactive
destroy()	Cleans up resources before unloading the applet	When applet is destroyed or removed from memory

15.6 DEVELOPING APPLETS AND TESTING

Developing applets involves creating a Java program that extends the java.applet.Applet or javax.swing.JApplet class (for Swing-based GUIs). These applets are embedded in HTML pages and executed within a Java-enabled browser or applet viewer. Testing applets ensures they function correctly across different platforms and browsers.

❖ **Steps for Developing an Applet**

1. **Set Up the Development Environment**:
   - Install the Java Development Kit (JDK) for compiling and running applets.
   - Use an Integrated Development Environment (IDE) like Eclipse, IntelliJ IDEA, or NetBeans for writing and debugging code.
2. **Create the Applet Class**:
   - Extend the Applet or JApplet class.
   - Override the life cycle methods (init(), start(), stop(), destroy()) as required.
3. **Write the Applet Code**:
   - Add user interface components (if needed) using AWT or Swing.
   - Implement event-handling mechanisms.
   - Use the applet's Graphics object for custom drawing, if applicable.
4. **Embed the Applet in an HTML Page**:
   - Use the <applet> or <object> tag in an HTML file to embed the applet.

Example HTML Code:

```
<html>
<body>
  <applet code="MyApplet.class" width="300" height="200">
  </applet>
</body>
</html>
```

5. **Compile the Applet**:
   - Compile the applet source code using the javac command:

     javac MyApplet.java
6. **Run the Applet**:
   - Use an applet viewer or browser to run the applet:

     appletviewer MyApplet.html

❖ **Testing Applets**

Testing applets involves verifying their behavior in various environments and under different conditions.

1. **Testing in Applet Viewer**:
    - Use the appletviewer tool provided with the JDK to run the applet in a sandboxed environment.
    - Check the applet's graphical interface, responsiveness, and functionality.
2. **Testing in Browsers**:
    - Embed the applet in an HTML page and test it in different browsers that support Java applets.
    - Verify browser-specific behaviors and compatibility issues.
3. **Testing for Security**:
    - Ensure the applet adheres to security restrictions, especially for network access and file handling.
    - Test signed and unsigned applets for permissions.
4. **Cross-Platform Testing**:
    - Run the applet on multiple operating systems (Windows, macOS, Linux) to verify compatibility.
    - Ensure consistent behavior across different screen resolutions and sizes.
5. **Stress and Performance Testing**:
    - Test the applet under high load conditions to ensure stability.
    - Measure performance to verify that the applet runs efficiently.

Sample Applet Code

Below is a simple applet that displays a message:

```
import java.applet.Applet;
import java.awt.Graphics;

public class MyApplet extends Applet {
   public void init() {
      System.out.println("Applet initialized");
   }
```

```
    public void start() {
        System.out.println("Applet started");
    }

    public void paint(Graphics g) {
        g.drawString("Hello, Applet!", 50, 50);
    }

    public void stop() {
        System.out.println("Applet stopped");
    }

    public void destroy() {
        System.out.println("Applet destroyed");
    }
}
```

Execution and Testing

1. **Compile the Applet**:

   javac MyApplet.java

2. **Create an HTML File**:

   html
   Copy code

```
<html>
<body>
    <applet code="MyApplet.class" width="300" height="200">
    </applet>
</body>
</html>
```

3. **Run Using Applet Viewer**:

   appletviewer MyApplet.html

4. **Browser Testing**:
   - Open the HTML file in a Java-enabled browser.
   - Verify the applet's graphical output and life cycle behavior.

15.7 PASSING PARAMETERS TO APPLETS

Java applets can accept parameters from the HTML page in which they are embedded. These parameters are specified using the <param> tag in the HTML file and can be accessed within the applet using the getParameter() method. This feature allows applets to be more dynamic and configurable based on user input or external settings.

Steps to Pass Parameters to an Applet

1. **Define Parameters in HTML**:
   - Use the <param> tag within the <applet> tag to specify the parameter name and value.

```
html
Copy code
<html>
<body>
  <applet code="MyApplet.class" width="300" height="200">
    <param name="param1" value="Hello, Applet!">
    <param name="param2" value="42">
  </applet>
</body>
</html>
```

2. **Access Parameters in the Applet**:
   - Use the getParameter(String name) method in the applet to retrieve parameter values.

Example:

```
import java.applet.Applet;
import java.awt.Graphics;

public class MyApplet extends Applet {
  private String message;
  private int number;

  public void init() {
    // Retrieve parameters from the HTML page
    message = getParameter("param1");
    String numStr = getParameter("param2");
```

```
        number = (numStr != null) ? Integer.parseInt(numStr) : 0;
    }
    public void paint(Graphics g) {
        g.drawString("Message: " + message, 20, 50);
        g.drawString("Number: " + number, 20, 70);
    }
}
```

3. **Compile and Test**:
   - Compile the applet:

     ```
     javac MyApplet.java
     ```

   - Test using the appletviewer:

     ```
     appletviewer applet.html
     ```

Example Explanation

HTML File:

- Defines two parameters:
  - param1 with the value "Hello, Applet!".
  - param2 with the value "42".

Applet Code:

- Retrieves the parameters using getParameter().
- The message parameter is stored as a String.
- The param2 parameter is converted to an integer using Integer.parseInt().

Output:

- The applet displays:

  ```
  makefile
  Copy code
  Message: Hello, Applet!
  Number: 42
  ```

Use Cases for Passing Parameters

1. **Dynamic Content**:
   - Applets can adapt their behavior based on parameter values (e.g., displaying user-specific messages or settings).
2. **Configuration Settings**:

- Parameters can provide initial values like color themes, fonts, or file paths.

3. **Interactivity**:
    - Parameters allow applets to take inputs from the embedding HTML page for dynamic interactions.

Points to Remember

- **Null Check**: Always check if the parameter is null before using it to avoid NullPointerException.
- **Data Types**: Parameters are retrieved as strings; you must convert them to the appropriate data type (e.g., integer or float) if necessary.
- **Security**: Applets cannot access system resources unless they are signed, so parameters cannot provide file paths or sensitive information unless the applet is trusted.

By passing parameters, applets become flexible and capable of serving diverse purposes in different contexts.

15.8 APPLET SECURITY ISSUES

Java applets are executed in a restricted environment called the "sandbox" to ensure the security of the user's system. This sandbox imposes strict limitations on what an applet can do, preventing it from accessing system resources that could compromise the user's security or privacy. Despite these measures, there are potential security issues and challenges associated with applets.

1. Sandbox Restrictions

- **Purpose**: The sandbox ensures that applets:
    - Cannot access the local file system (e.g., read or write files).
    - Cannot connect to any network location other than the server from which they were loaded.
    - Cannot execute native code or make system-level changes.
- **Limitation**: These restrictions may limit the functionality of legitimate applets.

2. Unsigned vs. Signed Applets

- **Unsigned Applets**:
    - Restricted to the sandbox and cannot perform sensitive operations.
    - Typically safe but limited in functionality.
- **Signed Applets**:
    - Signed with a digital certificate to verify their authenticity and request permission to bypass sandbox restrictions.

- Can access local files, network resources, and perform other privileged operations if the user grants permission.
- **Risk**: Malicious signed applets may exploit granted permissions to harm the system.

3. Common Security Risks

1. **Malicious Code Execution**:
   - A compromised or malicious applet may execute harmful operations if signed and granted permissions by the user.
2. **Network Exploitation**:
   - Applets can communicate with the server they were loaded from, potentially leading to unauthorized data transfer or Distributed Denial of Service (DDoS) attacks.
3. **Code Injection**:
   - Vulnerable applets can be exploited by attackers to execute arbitrary code on the user's machine.
4. **Man-in-the-Middle Attacks**:
   - Without proper encryption, applet data can be intercepted during transmission, leading to data breaches.
5. **Outdated Security Measures**:
   - Older Java versions may have unpatched vulnerabilities that can be exploited by attackers.

4. Security Features in Applets

To mitigate these risks, Java provides robust security mechanisms:

- **Code Signing**:
  - Applets can be signed with a trusted certificate to verify their origin and integrity.
- **Permissions**:
  - Users can allow or deny permissions to signed applets based on trust level.
- **Security Manager**:
  - The Java Security Manager enforces sandbox rules and prevents unauthorized operations.
- **Classloader**:
  - Applets are loaded through a custom class loader, ensuring that untrusted applet code cannot interfere with system-level classes.

- **Secure Communication**:
    - Applets can use SSL/TLS for secure data transmission.

5. Applet Security Policies

- **Default Policy**:
    - Unsigned applets are confined to the sandbox with minimal access to system resources.
- **Custom Policies**:
    - Developers or administrators can define custom security policies to allow or restrict specific applet operations.

6. Recommendations for Safe Applet Use

1. **Update Java Regularly**:
    - Use the latest version of Java to ensure all known vulnerabilities are patched.
2. **Avoid Untrusted Sources**:
    - Only run applets from trusted websites or developers.
3. **Verify Digital Signatures**:
    - Ensure that signed applets are verified by a trusted certificate authority.
4. **Educate Users**:
    - Inform users about granting permissions to applets and the associated risks.
5. **Minimize Dependencies on Applets**:
    - Use modern web technologies (like JavaScript and HTML5) as alternatives to applets, since browser support for applets is declining.

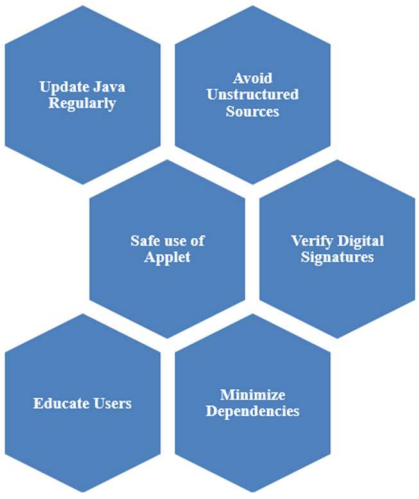


Fig 15.2 Recommendations for Safe Applet Use

15.9 SUMMARY

Applets have been a significant feature in Java, enabling dynamic and interactive content within web pages. Their **inheritance hierarchy** illustrates the robust object-oriented design of Java, while their **life cycle methods** (init(), start(), stop(), and destroy()) provide a structured way to manage applet execution. The **differences between applets and applications** highlight the unique, browser-based use case of applets compared to standalone programs. Developing and testing applets involves embedding them in HTML, configuring parameters, and ensuring compatibility across platforms. Passing parameters to applets makes them adaptable to different scenarios, enhancing interactivity and flexibility. However, **security issues**, such as sandbox restrictions and risks associated with signed applets, underline the importance of secure coding and user awareness. While applets have largely been replaced by modern technologies, understanding their architecture and principles offers valuable insights into Java programming and legacy systems.

15.10 TECHNICAL TERMS

Event , Event Sources, Listener, Adapter, Mouse Event, Action Event ,Key Event

15.11 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain the inheritance hierarchy of applets in Java. Provide an example to illustrate.
2. Discuss the life cycle of an applet with a detailed explanation of each method (init(), start(), stop(), destroy()).
3. Describe the process of developing and testing an applet, including writing HTML code to embed the applet.
4. How can parameters be passed to an applet? Write a program to demonstrate passing parameters using the <param> tag.
5. What are the common security issues in applets? How does Java address these issues to ensure applet safety?

Short Answer Questions:

1. What is an applet in Java?
2. List the four key methods of an applet's life cycle.
3. How do applets differ from standalone Java applications?
4. What is the purpose of the getParameter() method in applets?
5. Mention two common applet security restrictions.

15.12 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

AUTHOR: **Mrs. Appikatla Pushpa Latha**

LESSON- 16

GUI PROGRAMMING WITH JAVA

OBJECTIVES:

After going through this lesson, you will be able to

- Understand the AWT and Swing Frameworks:
- Differentiate Between Swing and AWT:
- Explore Hierarchies and Containers:
- Implement Swing Components:

STRUCTURE OF THE LESSION:

16.1 INTRODUCTION

Graphical User Interface (GUI) programming in Java is an essential aspect of creating user-friendly applications. Java provides two primary frameworks for GUI development: **Abstract Window Toolkit (AWT)** and **Swing**. The AWT class hierarchy forms the foundation, offering basic components and platform-dependent features. Swing, introduced later, builds upon AWT, offering a richer set of components, a **lightweight architecture**, and support for the **Model-View-Controller (MVC)** design pattern, enabling better separation of logic and UI. Swing includes a wide variety of components organized in a hierarchical structure, with **top-level containers** like JFrame, JApplet, JWindow, and JDialog, as well as **lightweight containers** like JPanel. Developers can build complex GUIs with Swing components such as JButton, JToggleButton, JCheckBox, JRadioButton, JLabel, JTextField, JTextArea, JList, JComboBox, and JMenu.

16.2 THE AWT CLASS HIERARCHY

The Abstract Window Toolkit (AWT) in Java provides a platform-independent framework for building graphical user interfaces (GUIs). It is part of Java's standard library and includes classes for windowing, event handling, and component management. The AWT class hierarchy is the foundation of Java GUI programming, comprising a variety of classes and interfaces that represent UI components, containers, and event listeners.

Key Classes in the AWT Hierarchy

1. **Component:**
   - The base class for all AWT components (e.g., buttons, text fields, labels).
   - Defines methods for managing size, position, and rendering.
2. **Container:**
   - A subclass of Component that can hold other components.
   - Examples include Panel, Window, and Frame.
3. **Panel:**
   - A generic container for grouping components.
   - Often used as a base for custom UI elements.

4. **Window:**
    - A top-level container without borders or a menu bar.
    - Extended by Frame and Dialog.
5. **Frame:**
    - A top-level window with a title and borders.
    - Commonly used as the main application window.
6. **Dialog:**
    - A pop-up window for user interaction, often modal (blocking other windows until dismissed).
7. **Button, Label, TextField, and Other Components:**
    - Specific UI elements derived from Component.
8. **Graphics and Canvas:**
    - Graphics is the base for rendering shapes and images.
    - Canvas is a blank drawable surface.
9. **LayoutManager:**
    - An interface for organizing components within a container.
    - Includes implementations like FlowLayout, BorderLayout, and GridLayout.

16.3 INTRODUCTION TO SWING

Swing is a Java-based GUI toolkit that provides a rich set of components for building robust, flexible, and platform-independent graphical user interfaces. Introduced as part of the Java Foundation Classes (JFC), Swing extends the Abstract Window Toolkit (AWT) by offering more advanced components, improved aesthetics, and a lightweight architecture. Unlike AWT, which relies on the native platform for rendering, Swing components are written entirely in Java, making them platform-independent and consistent in appearance across different operating systems.

Key Features of Swing

1. **Lightweight Components:** Swing components do not depend on the native operating system, making them more efficient and customizable than AWT.
2. **Rich Component Set:** Swing offers a comprehensive range of components, including JButton, JLabel, JTextField, JTable, JTree, JList, and many more.
3. **Pluggable Look-and-Feel:** Developers can change the appearance of Swing applications dynamically without altering the code, using built-in or custom look-and-feel themes.
4. **Event Handling:** Swing inherits AWT's event delegation model, providing a robust mechanism to handle user actions like button clicks, mouse movements, and keyboard events.
5. **Customizable Components:** Swing allows developers to extend and modify components to suit application-specific requirements.
6. **MVC Architecture:** Swing follows the Model-View-Controller (MVC) design pattern, separating data handling (Model) from the visual representation (View) and user interaction logic (Controller).
7. **Double Buffering:** To ensure smoother graphics, Swing uses double buffering, reducing flickering during animations or rapid updates.

Advantages of Swing

1. **Platform Independence:** Swing components are rendered by Java, ensuring consistent behavior and appearance across all platforms.
2. **Advanced Graphics:** Provides a rich graphics API for creating custom components and high-quality visuals.
3. **Extensibility:** Developers can extend existing Swing classes or create their own components.

4. **Built-in Support for Layout Management:** Swing includes various layout managers like BorderLayout, FlowLayout, GridLayout, and BoxLayout for efficient GUI design.

Limitations of Swing

1. **Performance Overhead:** Swing's rich features can lead to higher memory usage and slower performance compared to lightweight GUI frameworks.
2. **Complexity:** Swing applications can become complex to manage, especially for larger projects.
3. **Declining Usage:** With modern alternatives like JavaFX, the adoption of Swing has decreased in recent years.

Applications of Swing

- Desktop applications like text editors, media players, and database front-ends.
- Educational tools and simulations.
- Prototyping UI designs.

16.4 SWING VS. AWT

Table 16.1 Swing vs. AWT

Feature	AWT	Swing
Type of Components	Heavyweight components (depend on the OS)	Lightweight components (entirely written in Java)
Rendering	Relies on native OS components for rendering	Java-based rendering (platform-independent)
Look and Feel	Native look and feel (varies with OS)	Pluggable Look and Feel (PLAF) – customizable styles
Performance	Faster performance due to native rendering	Slower performance due to Java-based rendering

Components	Limited set of components (e.g., Button, TextField, Label)	Richer set of components (e.g., JTable, JTree, JComboBox)
Layout Management	Basic layout managers (e.g., FlowLayout, BorderLayout)	Advanced layout managers with more flexibility (e.g., GridLayout, BoxLayout)
Platform Dependency	Platform-dependent, may vary across OS	Platform-independent, consistent across OS
Event Handling	AWT-based event handling	Swing inherits AWT's event handling model with improvements
Customization	Limited customization and flexibility	High level of customization and flexibility
Graphics	Limited graphics capabilities	Enhanced graphics capabilities (e.g., custom painting)
Integration	Can integrate with Swing, but not as flexible	Fully self-contained, no dependency on AWT for most functionality
Examples	Button, Label, TextField, Checkbox, Panel	JButton, JLabel, JTextField, JComboBox, JTable

16.5 MVC ARCHITECTURE

The Model-View-Controller (MVC) architecture is a design pattern used to separate the concerns in an application, making it easier to manage and scale. It is widely used in GUI-based applications and provides a structured approach to designing the user interface (UI) by dividing it into three interconnected components:

1. **Model:**
    - The Model represents the data and business logic of the application. It directly manages the data, logic, and rules of the application.
    - Responsibilities:
        - Holds the application data and state.
        - Notifies the View of any changes to the data.
        - Performs all the necessary operations on the data (such as calculations, querying the database, etc.).
    - Example: A class that holds user data (like a Student class or a BankAccount class).
2. **View:**
    - The View is responsible for the presentation layer. It displays the data from the Model to the user and sends user commands to the Controller.
    - Responsibilities:
        - Renders the user interface (UI) elements (buttons, labels, tables, etc.).
        - Listens for any changes in the Model and updates itself accordingly.
        - Relays user input to the Controller.
    - Example: A window or screen showing data such as a table of user records or a form for input.
3. **Controller:**
    - The Controller acts as an intermediary between the Model and View. It listens to user input (through the View) and updates the Model accordingly.
    - Responsibilities:
        - Handles user inputs (like mouse clicks, keyboard actions).

- Updates the Model based on user actions.
- Requests updates from the Model and passes the results to the View.

- Example: A class or method that reacts to button clicks or other user actions and updates the Model accordingly (e.g., adding a new user to the database).

How MVC Works Together:

1. **User Interaction:**
   - The user interacts with the View (e.g., clicking a button or entering data into a text field).
2. **Controller Response:**
   - The View sends the user's actions to the Controller. The Controller then processes the input, typically making changes to the Model.
3. **Model Update:**
   - The Model reflects the changes (e.g., updating data or processing logic) and notifies the View of the update.
4. **View Update:**
   - The View queries the Model and updates the UI with the new data, reflecting the changes made by the user.

Advantages of MVC Architecture:

1. **Separation of Concerns:**
   - The MVC pattern ensures that the application logic, UI, and user input handling are separated into distinct components, making the application easier to maintain and extend.
2. **Modularity:**
   - Each component (Model, View, and Controller) can be modified independently, which allows for easier updates and customization.

3. **Code Reusability:**
    - The separation allows different Views to be associated with the same Model, facilitating code reuse.
4. **Easier Maintenance:**
    - Changes in the data model or business logic (Model) don't require modifications to the View or Controller, making it easier to maintain the application.

16.6 HIERARCHY FOR SWING COMPONENTS

Swing components are part of the Java Foundation Classes (JFC), and they follow a specific hierarchy that allows for easy manipulation and extension of user interface elements. The Swing component hierarchy is built on top of AWT components but introduces a more flexible, lightweight framework for creating GUIs in Java. Below is the hierarchy of Swing components, which illustrates the inheritance structure and relationships between key components.

1. **JComponent:**
    - JComponent is the base class for all Swing components that have a graphical user interface. It extends java.awt.Component and provides many essential methods like setBackground(), setForeground(), setFont(), setSize(), and setVisible(), among others.
    - Direct subclasses of JComponent represent the majority of Swing UI components.
2. **JButton, JLabel, JTextField, JTextArea:**
    - These are commonly used Swing components for user interaction:
        - JButton: A clickable button.
        - JLabel: Displays text or images.
        - JTextField: A single-line text input field.
        - JTextArea: A multi-line text input area.

3. **JToggleButton, JCheckBox, JRadioButton:**
    - These components represent toggle-style input elements:
        - JToggleButton: A button that switches between on and off states.
        - JCheckBox: A box that can be checked or unchecked.
        - JRadioButton: A radio button used in groups where only one button can be selected at a time.
4. **JComboBox, JList, JTree:**
    - These components are used for displaying and selecting data:
        - JComboBox: A drop-down list allowing selection from a set of options.
        - JList: Displays a list of items.
        - JTree: A hierarchical tree structure for displaying data in a tree format.
5. **Containers:**
    - Containers are special types of components that hold other components. Swing provides various types of containers:
        - JPanel: A generic container used for organizing components.
        - JScrollPane: A container that provides a scrollable view for other components, such as text areas or lists.
        - JFrame: A top-level container used to represent a window.
        - JDialog: A pop-up dialog window for secondary interactions.
        - JWindow: A top-level window without borders or decorations.
        - JApplet: A container for applets (although applets are less common in modern Java applications).

6. **Menus and Other Controls:**
    - Swing also includes several other UI controls for interaction, such as:
        - JMenu: A menu component used in menus.
        - JMenuItem: A menu item inside a JMenu.
        - JRadioButtonMenuItem: A radio button inside a menu.
        - JTextPane: A component that can display styled text, allowing text formatting like bold, italic, and color.

16.7 CONTAINERS IN JAVA SWING

In Java Swing, containers are special components that can hold other components (like buttons, text fields, etc.) and are responsible for organizing and managing the layout of these components. There are two main categories of containers in Swing: Top-level containers and Lightweight containers. Below is a detailed explanation of each:

❖ **Top-Level Containers**

Top-level containers are the primary containers in Swing that form the foundation of the application window. They are used to create the outer shell or window for the application.

➢ **JFrame:**

- JFrame is one of the most commonly used top-level containers in Swing. It represents a standard window in a desktop application.
- Features:
    - Can have a title bar, minimize, maximize, and close buttons.
    - Typically used for standalone desktop applications.
    - Supports menus, toolbars, and other UI components.
- Example Usage:

```
JFrame frame = new JFrame("My First Frame");
frame.setSize(400, 300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
frame.setVisible(true);
```

- **JApplet (Deprecated):**
  - JApplet was used for embedding small Java applications (applets) inside web browsers. However, it has been deprecated and is no longer commonly used.
  - Features:
    - Initially used to provide interactive content in web browsers.
    - Can be embedded in web pages but requires the browser to support Java Applets.
- **JWindow:**
  - JWindow is a top-level container that creates an undecorated window. Unlike JFrame, JWindow has no title bar or any other window decorations.
  - Features:
    - Useful for creating splash screens or other pop-up windows without borders.
    - Can be used as a floating, standalone window with no typical window controls.
  - **Example Usage:**

```
JWindow window = new JWindow();
window.setSize(200, 100);
window.setVisible(true);
```

- **JDialog:**
  - JDialog is used for creating pop-up dialog windows, usually for alerts, confirmations, or user input. It is a modal or non-modal window.
  - Features:
    - Can be used as a modal dialog (blocks interaction with other windows until it is closed) or non-modal (allows interaction with other windows).
    - Useful for getting user input or displaying messages.

```
JDialog dialog = new JDialog(frame, "Dialog Example", true); // Modal dialog
dialog.setSize(200, 100);
dialog.setVisible(true);
```

❖ **Lightweight Containers**

Lightweight containers are containers that do not rely on the operating system's native windowing system. These containers are drawn purely in Java and allow for more flexibility and consistency across different platforms.

➢ **JPanel:**

- JPanel is one of the most commonly used lightweight containers in Swing. It is a general-purpose container that can hold and organize components.
- Features:
  - Used to group multiple components together inside a larger container (like JFrame or JDialog).
  - Can be used with layout managers to organize the components it holds.
  - Does not have window decorations or a title bar (unlike JFrame or JDialog).

```
JPanel panel = new JPanel();
JButton button = new JButton("Click Me");
panel.add(button);
frame.add(panel);
```

Table 16.2 Comparison of Top-Level and Lightweight Containers

Feature	**Top-Level Containers (JFrame, JApplet, JWindow, JDialog)**	**Lightweight Containers (JPanel)**
Purpose	Used as the outer container	Used to organize and group

	or window for the application.	components within other containers.
Window Decorations	Can have title bars, close buttons, etc. (e.g., JFrame, JWindow)	No window decorations.
Visibility	Can be made visible as standalone windows.	Needs to be added to a top-level container to be visible.
Usage	Typically used for main windows, dialog boxes, and splash screens.	Used for grouping and laying out components inside other containers.
Examples	JFrame, JDialog, JWindow	JPanel

- Top-level containers are used to create primary windows and dialog boxes, forming the main structure of a GUI application.
- Lightweight containers, such as JPanel, help organize the layout and grouping of components within those top-level windows. Together, these containers help build a robust and organized graphical user interface in Java Swing.

16.8 A SIMPLE SWING APPLICATION

Here's a simple Java Swing application that demonstrates the usage of the listed Swing components. The application includes a basic graphical user interface (GUI) with interactive elements such as buttons, toggles, checkboxes, radio buttons, and more.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SwingDemoApp {
    public static void main(String[] args) {
        // Create the main JFrame
```

```
JFrame frame = new JFrame("Swing Components Demo");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(500, 500);
frame.setLayout(new BorderLayout());

// Create a JPanel for the components
JPanel panel = new JPanel();
panel.setLayout(new GridLayout(0, 2, 10, 10));

// JLabel
JLabel label = new JLabel("Label: ");
panel.add(label);

// JTextField
JTextField textField = new JTextField();
panel.add(textField);

// JButton
JButton button = new JButton("Button");
panel.add(button);

// JToggleButton
JToggleButton toggleButton = new JToggleButton("Toggle Button");
panel.add(toggleButton);

// JCheckBox
JCheckBox checkBox = new JCheckBox("Checkbox");
panel.add(checkBox);

// JRadioButton
JRadioButton radioButton = new JRadioButton("Radio Button");
panel.add(radioButton);

// JTextArea
```

```
        JTextArea textArea = new JTextArea(5, 20);
        JScrollPane textAreaScroll = new JScrollPane(textArea);
        panel.add(new JLabel("TextArea:"));
        panel.add(textAreaScroll);

        // JList
        DefaultListModel<String> listModel = new DefaultListModel<>();
        listModel.addElement("Item 1");
        listModel.addElement("Item 2");
        listModel.addElement("Item 3");
        JList<String> list = new JList<>(listModel);
        JScrollPane listScroll = new JScrollPane(list);
        panel.add(new JLabel("List:"));
        panel.add(listScroll);

        // JComboBox
        JComboBox<String> comboBox = new JComboBox<>(new String[]{"Option 1",
"Option 2", "Option 3"});
        panel.add(new JLabel("ComboBox:"));
        panel.add(comboBox);

        // JMenu
        JMenuBar menuBar = new JMenuBar();
        JMenu menu = new JMenu("Menu");
        JMenuItem menuItem1 = new JMenuItem("Menu Item 1");
        JMenuItem menuItem2 = new JMenuItem("Menu Item 2");
        menu.add(menuItem1);
        menu.add(menuItem2);
        menuBar.add(menu);
        frame.setJMenuBar(menuBar);

        // ActionListener for button
        button.addActionListener(e -> JOptionPane.showMessageDialog(frame, "Button
clicked!"));
```

```
        // Add components to the frame
        frame.add(panel, BorderLayout.CENTER);

        // Make the frame visible
        frame.setVisible(true);
    }
}
```

16.9 SUMMARY

GUI Programming with Java focuses on creating graphical user interfaces using Java's Abstract Window Toolkit (AWT) and Swing libraries. The AWT class hierarchy provides a foundation for GUI components but is platform-dependent and less flexible. Swing is an extension of AWT, offering lightweight, platform-independent, and more versatile components. Unlike AWT, Swing is built entirely in Java and follows the Model-View-Controller (MVC) architecture, separating data (model), user interface (view), and logic (controller).

Swing components are organized in a hierarchy, starting with containers, which can hold other components. Top-level containers include JFrame (main window), JApplet (applet window), JWindow (borderless window), and JDialog (popup dialog). Lightweight containers like JPanel are used for grouping components. A simple Swing application showcases these components, demonstrating their ease of use and ability to create intuitive, responsive GUIs.

16.10 TECHNICAL TERMS

GUI, AWT, Swing, Model, View, Controller, JPanel, JFrame, JButton.

16.11 SELF ASSESSMENT QUESTIONS

Essay questions:

1. Explain the AWT class hierarchy and its significance in GUI programming.
2. Discuss the advantages of Swing over AWT and how it addresses AWT's limitations.
3. Describe the hierarchy of Swing components and differentiate between top-level containers and lightweight containers.
4. Write a simple Swing application that demonstrates the use of JButton, JCheckBox, JRadioButton, JLabel, and JTextField.

5. How does the MVC architecture enhance the functionality of Swing components? Illustrate with an example.

Short Answer Questions:

1. What is the primary difference between AWT and Swing in Java?
2. Define the MVC architecture in the context of Swing components.
3. Name the top-level containers available in Swing.
4. What is a lightweight container in Swing, and give an example.
5. List any four Swing components and their primary uses.

16 .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

AUTHOR: **Mrs. Appikatla Pushpa Latha**

LESSON- 17

JAVA'S GRAPHICS CAPABILITIES

OBJECTIVES:

After going through this lesson, you will be able to

- Understand Graphics Contexts and Graphics Objects
- Explore Color and Font Control
- Familiarize with Layout Management
- Design and Implement Graphical Applications

STRUCTURE OF THE LESSION:

17.1 INTRODUCTION

Java's graphics capabilities enable developers to create visually engaging applications by leveraging the Graphics class and its associated tools. The Graphics context serves as an environment where drawing operations occur, providing methods to render shapes, text, and images. The Graphics object acts as an interface to this context, offering tools to customize drawings. Developers can control visual aspects such as color using the Color class, which allows for setting background, foreground, and drawing colors, and font customization with the Font class to display styled text. Java's graphics framework supports rendering basic shapes like lines, rectangles, and ovals, as well as arcs for creating more complex figures. These tools

collectively empower developers to create detailed and expressive graphical elements in applications.

In addition to its drawing capabilities, Java simplifies GUI design with layout management, ensuring components are arranged consistently across different screen sizes and resolutions. Layout managers, such as BorderLayout, organize components around a central region with north, south, east, and west areas. GridLayout divides the container into a grid of rows and columns, while FlowLayout arranges components sequentially, respecting their preferred sizes. BoxLayout offers flexibility for stacking components horizontally or vertically. By combining robust graphics tools with efficient layout management, Java provides a comprehensive platform for building both functional and aesthetically pleasing graphical user interfaces.

17.2 JAVA'S GRAPHICS CAPABILITIES

Java provides a powerful and flexible set of tools for graphics programming through its Java 2D API, which is part of the javax.swing and java.awt packages. These tools enable you to create, manipulate, and display graphics in Java applications. Graphics programming is essential for building interactive applications with custom visual elements, including shapes, colors, text, and images.

Graphics Contexts and Graphics Objects

- **Graphics Class**: The Graphics class in Java provides the basic methods for drawing shapes, text, and images. It is the abstract base class for all graphics contexts. The actual drawing is handled by the Graphics2D class, which extends Graphics and provides more advanced drawing capabilities, such as gradient fills, complex shapes, and advanced transformations.
- **Graphics Context**: A graphics context is an environment that holds the information related to drawing. It includes the current drawing attributes such as color, font, and stroke (line width). Each time you invoke drawing methods, they use the current graphics context to render the output.

17.3 GRAPHICS CONTEXT

The Graphics context serves as an environment where drawing operations occur, providing methods to render shapes, text, and images. The Graphics object acts as an interface to this context, offering tools to customize drawings. Developers can control visual aspects such as color using the Color class, which allows for setting background, foreground, and drawing colors, and font customization with the Font class to display styled text. Java's graphics framework supports rendering basic shapes like lines, rectangles, and ovals, as well as arcs for

creating more complex figures. These tools collectively empower developers to create detailed and expressive graphical elements in applications.

1. **Color Control**:
    - Java provides the Color class to manage colors in graphics. You can set colors using predefined colors (Color.RED, Color.BLUE, etc.) or define custom colors using RGB values.
    - Example:

```
Color myColor = new Color(255, 0, 0); // Red
g.setColor(myColor);
```

2. **Font Control**:
    - The Font class enables font customization. You can choose font style (bold, italic) and size.
    - Example:

```
Font myFont = new Font("Arial", Font.ITALIC, 20);
g.setFont(myFont);
g.drawString("Styled Text", 50, 250);
```

3. **Drawing Shapes**:
    - Java's Graphics API allows for drawing basic geometric shapes like lines, rectangles, circles, and ovals. You can use both outlines and filled shapes.
4. **Drawing Text**:
    - The drawString() method allows you to render text on the screen at specific coordinates.
    - Example:

```
g.drawString("Hello, World!", 100, 100);
```

Advanced Graphics Capabilities

1. **Graphics2D**:
    - Graphics2D extends Graphics and provides more control over the rendering process, including advanced features like transformations, gradients, and complex shapes.
    - Example of setting a gradient color:

```
GradientPaint gradient = new GradientPaint(0, 0, Color.RED, 100, 100, Color.BLUE);

g2d.setPaint(gradient);

g2d.fillRect(50, 50, 200, 100);
```

2. **Affine Transformations**:
   - You can rotate, scale, or shear graphics using AffineTransform. This allows for complex animations and transformations of shapes.
   - Example:

```
AffineTransform transform = new AffineTransform();

transform.rotate(Math.toRadians(45), 100, 100); // Rotate 45 degrees around (100, 100)

g2d.setTransform(transform);

g2d.fillRect(50, 50, 100, 100);
```

3. **Clipping**:
   - Java allows clipping the drawing area with a rectangular or custom-shaped region. Only the parts inside the clipping area are drawn.
   - Example:

```
g2d.setClip(50, 50, 200, 200); // Clip to a rectangle
```

17.4 GRAPHICS OBJECT

The Graphics class in Java is the cornerstone of its graphics capabilities, providing a powerful API for rendering 2D shapes, text, and images. It acts as an interface to a **graphics context**, encapsulating details about the drawing surface and rendering attributes such as color, font, and clipping area. Developers typically obtain a Graphics object by overriding the paint() or paintComponent() methods in Swing components like JPanel, which automatically pass the Graphics object as a parameter. With methods like drawLine(), drawRect(), and drawOval(), the Graphics class facilitates the drawing of shapes, while fillRect() and fillOval() render filled shapes. For displaying text, methods like drawString() allow developers to incorporate dynamic or static text into their graphical interfaces.

In addition to basic drawing operations, the Graphics class supports customizations through methods like setColor() for setting the drawing color and setFont() for defining text styles. While the Graphics class is versatile, the Graphics2D class, which extends it, offers enhanced

features for advanced rendering such as transformations, strokes, and anti-aliasing. Together, these classes form the foundation of Java's 2D graphics system, enabling developers to create everything from simple drawings to intricate, interactive graphical user interfaces.

```
import javax.swing.*;
import java.awt.*;

public class GraphicsExample extends JPanel {
   @Override
   protected void paintComponent(Graphics g) {
      super.paintComponent(g); // Ensures proper rendering of the panel
            // Set color for drawing
      g.setColor(Color.BLUE);
            // Draw a line
      g.drawLine(50, 50, 200, 50);
            // Draw a rectangle
      g.drawRect(50, 70, 150, 100);
            // Fill a rectangle
      g.setColor(Color.RED);
      g.fillRect(220, 70, 150, 100);
            // Draw an oval
      g.setColor(Color.GREEN);
      g.drawOval(50, 200, 100, 50);
            // Fill an oval
      g.setColor(Color.ORANGE);
      g.fillOval(200, 200, 100, 50);

      // Draw an arc
      g.setColor(Color.MAGENTA);
      g.drawArc(50, 300, 100, 100, 0, 180);

      // Fill an arc
      g.setColor(Color.CYAN);
      g.fillArc(200, 300, 100, 100, 0, 180);
```

```
        // Draw text
        g.setColor(Color.BLACK);
        g.setFont(new Font("Serif", Font.BOLD, 16));
        g.drawString("Graphics Class Example", 50, 450);
    }

    public static void main(String[] args) {
        // Create a JFrame
        JFrame frame = new JFrame("Graphics Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 500);

        // Add the custom JPanel
        GraphicsExample panel = new GraphicsExample();
        frame.add(panel);

        // Make the frame visible
        frame.setVisible(true);
    }
}
```

- **Custom Panel (GraphicsExample)**:
  - The paintComponent(Graphics g) method is overridden to perform custom drawing.
  - The super.paintComponent(g) call ensures that the panel is properly rendered before drawing.
- **Drawing Shapes**:
  - Methods like drawLine(), drawRect(), drawOval(), and drawArc() are used for outlines.
  - fillRect(), fillOval(), and fillArc() are used to draw filled shapes.
- **Customizing Colors and Fonts**:
  - The setColor() method sets the drawing color.
  - The setFont() method customizes the font for drawing text.
- **JFrame Setup**:

- A JFrame is used to host the custom panel where graphics are drawn.

17.5 LAYOUT MANAGEMENT

In Java, Layout Managers are used to define the layout and arrangement of components within a container, making it easier to design flexible, responsive graphical user interfaces (GUIs). Layout managers automatically manage the size and position of components, preventing components from overlapping or being placed improperly. This ensures that the user interface looks well-structured across different screen sizes and resolutions.

Java provides several built-in layout managers, each serving a specific purpose. These include BorderLayout, GridLayout, FlowLayout, and BoxLayout. Below is a detailed explanation of these layout managers and their use cases:

❖ BorderLayout

- The **BorderLayout** manager divides the container into five regions: **North**, **South**, **East**, **West**, and **Center**. Components added to the **North** and **South** regions are placed horizontally, while components added to **East** and **West** are placed vertically. The **Center** region occupies the remaining available space and can expand or contract as the container resizes.
- **Use Case**: Useful for arranging components in a single panel with clearly defined areas, like menus, toolbars, or status bars. Often used in top-level containers (like **JFrame**).

Example:

```
import java.awt.*;

import javax.swing.*;

public class BorderLayoutExample {

   public static void main(String[] args) {

      JFrame frame = new JFrame("BorderLayout Example");

      frame.setSize(400, 300);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.setLayout(new BorderLayout());

      frame.add(new JButton("North"), BorderLayout.NORTH);

      frame.add(new JButton("South"), BorderLayout.SOUTH);

      frame.add(new JButton("East"), BorderLayout.EAST);

      frame.add(new JButton("West"), BorderLayout.WEST);
```

```
        frame.add(new JButton("Center"), BorderLayout.CENTER);
        frame.setVisible(true);
    }
}
```

In the above example, the buttons are placed in the **North**, **South**, **East**, **West**, and **Center** regions of the BorderLayout.

❖ **GridLayout**

- The **GridLayout** manager arranges components in a grid with a specified number of rows and columns. Each cell in the grid has the same size. The components are placed sequentially in the grid, from left to right and top to bottom.
- **Use Case**: Ideal for situations where you need to display components in a uniform grid, such as calculators, form layouts, or grid-based views.

Example:

```
import java.awt.*;
import javax.swing.*;
public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridLayout Example");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new GridLayout(2, 2));  // 2 rows, 2 columns
        frame.add(new JButton("Button 1"));
        frame.add(new JButton("Button 2"));
        frame.add(new JButton("Button 3"));
        frame.add(new JButton("Button 4"));
        frame.setVisible(true);
    }
}
```

In this example, the GridLayout manager arranges the buttons in a grid with 2 rows and 2 columns. Each button takes up one cell in the grid.

❖ **FlowLayout**

- The **FlowLayout** manager arranges components in a single row, one after the other. When the row is full, it wraps to the next line. You can control the alignment and the horizontal/vertical gaps between components. Components are added sequentially, and the layout adjusts based on the available space.
- **Use Case**: Useful for form layouts or when you want components to flow in a single line or wrap around based on available space, such as toolbars or simple horizontal menus.

Example:

```
import java.awt.*;

import javax.swing.*;

public class FlowLayoutExample {

   public static void main(String[] args) {

      JFrame frame = new JFrame("FlowLayout Example");

      frame.setSize(400, 100);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);



      frame.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20)); // Center alignment, 20px gap

      frame.add(new JButton("Button 1"));

      frame.add(new JButton("Button 2"));

      frame.add(new JButton("Button 3"));

      frame.setVisible(true);

   }

}
```

The FlowLayout manager arranges the buttons sequentially in a single row. If the window is resized, the components will wrap to the next line automatically.

❖ **BoxLayout**

- The **BoxLayout** manager arranges components either vertically or horizontally. The components are aligned and stretched in a single line, either in a row or column. The BoxLayout is flexible in handling different component sizes and layouts.
- **Use Case**: Ideal when you need components to be aligned in a single direction (either horizontally or vertically), such as in forms, toolbars, or simple vertical/horizontal layouts.

Example:

```
import java.awt.*;

import javax.swing.*;

public class BoxLayoutExample {

   public static void main(String[] args) {

      JFrame frame = new JFrame("BoxLayout Example");

      frame.setSize(300, 200);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Vertical BoxLayout

      frame.setLayout(new BoxLayout(frame.getContentPane(), BoxLayout.Y_AXIS));

      frame.add(new JButton("Button 1"));

      frame.add(new JButton("Button 2"));

      frame.add(new JButton("Button 3"));


      frame.setVisible(true);

   }

}
```

In this example, the BoxLayout is set to arrange the buttons vertically (BoxLayout.Y_AXIS). The buttons are stacked on top of each other.

17.1 Summary of Layout Manager Types

Layout Manager	Description	Use Case
BorderLayout	Divides the container into five regions: North, South, East, West, and Center.	Ideal for top-level containers like frames or windows.

GridLayout	Arranges components in a grid with rows and columns of equal size.	Perfect for grid-based layouts like calculators or forms.
FlowLayout	Components are arranged in a single row that wraps to the next line when necessary.	Suitable for toolbars, menus, or simple sequential layouts.
BoxLayout	Arranges components in a single direction (either vertical or horizontal).	Great for forms, vertical or horizontal toolbars, or stacked components.

Each layout manager in Java serves a unique purpose and is suitable for different types of user interface designs. By choosing the appropriate layout manager, you can create flexible and responsive layouts that adjust automatically to the size of the container and its components. Whether you're building a simple form or a complex GUI application, understanding and using the right layout manager is key to designing an effective and user-friendly interface.

17.6 SUMMARY

Java's graphics capabilities provide a powerful framework for creating visually appealing applications, centered around the Graphics class. This class provides a **graphics context**, an environment where rendering operations like drawing shapes, text, and images take place. The Graphics object, typically obtained through the paint() or paintComponent() methods, allows developers to draw lines, rectangles, ovals, arcs, and more using methods like drawLine(), drawRect(), and drawOval(). Additionally, Java offers control over visual attributes through the Color and Font classes, enabling developers to customize text styles and the colors used for rendering. The introduction of the Graphics2D class extends these capabilities by supporting advanced features like transformations, anti-aliasing, and compositing.

Java also simplifies GUI design with its **layout management system**, which ensures consistent component arrangement across different screen sizes. Layout managers like **BorderLayout** organize components into regions (north, south, east, west, and center), while **GridLayout** divides the container into equal cells. **FlowLayout** arranges components sequentially in a row, and **BoxLayout** stacks them horizontally or vertically. These layout managers eliminate the need for manual component placement, allowing developers to focus on the application's functionality and aesthetics. Combined with its robust graphics tools, Java provides a comprehensive platform for developing dynamic and user-friendly graphical interfaces.

17.7 TECHNICAL TERMS

Graphic Object, Graphic Class, Graphic Context, Color Control, Font Control, Layout Manager, Border, and Grid.

17.8 SELF ASSESSMENT QUESTIONS

Essay questions:

- Explain the concept of **Graphics Context** and how the Graphics object is used to render shapes and text in Java applications.
- Describe the process of **color control** in Java's graphics system and how the Color class is used to modify drawing attributes.
- Discuss the role of **font control** in Java's graphics capabilities and how the Font class allows developers to customize text style and size.
- Provide an example demonstrating the drawing of lines, rectangles, ovals, and arcs in Java using the Graphics class. Explain how each shape is drawn and filled.
- Discuss the different **layout managers** in Java, such as **BorderLayout**, **GridLayout**, **FlowLayout**, and **BoxLayout**, and explain when to use each layout manager in GUI design.

Short Answer Questions:

- What is the purpose of the Graphics class in Java?
- How do you control the color of shapes and text in Java graphics?
- What is the difference between drawRect() and fillRect() methods in Java?
- Name the four layout manager types in Java and briefly describe their function.
- What is the use of Graphics2D class in Java?

17.9 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

AUTHOR: **Mrs. Appikatla Pushpa Latha**