UNIT-I:

Fundamentals of Object-Oriented Programming: Basic Concepts of Object Oriented Programming (OOP), Benefits and Applications of OOP. Java Evolution: Java Features, Difference between Java, C and C++, Java and Internet, Java Environment. Overview of Java Language: Introduction to Simple Java Program, Use of Comments and Math function, Application of two classes, Java Program Structure, Java Tokens and statements, Implementing Java program and JVM, Command Line Arguments

UNIT-II:

Constants, Variables and Data Types: Constants, Variables, Data Types, Declaration of Variables, Giving values to Variables, Symbolic Constants, Typecasting. Operators & Expressions: Arithmetic operators, Relational operators, Logical operators, Assignment operators, Increment & Decrement operators, conditional operators, Bitwise operators, Arithmetic Expressions, Evaluation of Expressions, Type Conversions in Expressions, Operator Precedence & Associatively. Decision Making, Branching & Looping: Decision Making with Control Statements, Looping Statements, Jump in loops, Labelled loops.

UNIT-III:

Classes, Objects and Methods: Defining Class, Methods Declaration, Constructors, Methods Overloading, Overriding Methods, Inheritance. Arrays, Strings and Vectors: 1D arrays, Creating an Array, 2D arrays, Strings, Vectors, Wrapper Classes, Enumerated Types. Inheritance: Defining, extending classes, and Implementing Interfaces. Multiple inheritance and polymorphism.

UNIT IV:

Packages: Basics of packages, System packages, Creating and accessing packages, Creating user defined packages, Adding class to a package. Exception Handling: Using the main keywords of exception handling: try, catch, throw, throws and finally; Nested try, Multiple catch statements, Creating user defined exceptions.

Introduction to Java

1.1 Features of Java .....
1.2 Java Byte code .....
1.3 Structure of Java Program .....

1. ***Language Fundamentals***
   - *2.1 Tokens* .....
     - *2.1.1 Comments* .....
     - *2.1.2 Identifiers* .....
     - *2.1.3 Keywords* .....
   - *2.2 Data Types* .....
   - *2.3 Variables* .....
     - *2.3.1 Reference Variables* .....
     - *2.3.2 Local Variables* .....
     - *2.3.3 Static Variables* .....
     - *2.3.4 Final Variables* .....
   - *2.4 Command Line Arguments* .....
   - *2.5 Operators* .....
   - *2.6 Expressions* .....
   - *2.7 Type Conversion and Casting* .....
   - *2.8 Control Statements* .....
2. ***Object-Oriented Programming Basics***
   - *3.1 Classes* .....
   - *3.2 Objects* .....
   - *3.3 Constructors* .....
   - *3.4 Methods* .....
     - *3.4.1 Types of Methods* .....
   - *3.5 Access Control* .....
     - *3.5.1 Public* .....
     - *3.5.2 Private* .....
     - *3.5.3 Protected* .....
     - *3.5.4 Default* .....
   - *3.6 this Keyword* .....
   - *3.7 Garbage Collection* .....
   - *3.8 Nested Classes* .....
   - *3.9 Inner Classes* .....
   - *3.10 String* .....
   - *3.11 String Buffer* .....
   - *3.12 String Builder* .....

1. ***Inheritance***
   - *1.1 Inheritance Concept* .....

1.2 Types of Inheritance
1.3 Member Access Rules
1.4 super Keyword
1.5 final Keyword

2. ***Abstract Classes and Polymorphism***
2.1 Abstract Classes
2.2 Abstract Methods
2.3 Polymorphism
2.3.1 Method Overriding (Run-time Polymorphism)
2.3.2 Method Overloading (Compile-time Polymorphism)
2.4 Constructor Overloading

3. ***Interfaces***
3.1 Defining an Interface
3.2 Implementing Interfaces
3.3 Extending Interface

4. ***Packages***
4.1 Inbuilt Packages
4.2 User Defined Packages
4.2.1 Defining a Package
4.2.2 Creating a Package
4.2.3 Accessing a Package
4.3 Importing Packages

5. ***Exception Handling***
5.1 Exception Handling
5.1.1 try Block
5.1.2 catch Block
5.1.3 throw Keyword
5.1.4 throws Keyword
5.1.5 finally Block
5.2 Exception Hierarchy
5.3 Exception vs Error
5.4 Classification of Exceptions
5.4.1 Checked Exceptions
5.4.2 Unchecked Exceptions
5.5 Re-throwing Exceptions
5.6 Types of Exception
5.6.1 Built-in Exceptions
5.6.2 Custom Exceptions

1. ***Multithreading***
1.1 Java Thread Model
1.2 Thread
1.3 Thread Life Cycle

1.4 Main Thread
1.5 Creating Threads
1.5.1 Implementing Runnable Interface
1.5.2 Extending Thread Class
1.6 Multiple Threads
1.7 Thread Priorities
1.8 Synchronization

2. ***Collections Framework***
2.1 Overview of Java Collections Framework
2.2 Collection Interface
2.3 List Interface
2.3.1 Array List
2.3.2 Linked List
2.4 Set Interface
2.4.1 Hash Set
2.4.2 Tree Set
2.5 Iterator
2.6 Map Interface

3. ***Utility Classes***
3.1 String Tokenizer
3.2 Date
3.3 Calendar
3.4 Gregorian Calendar
3.5 Scanner

4.Java Input and Output
4.1 java.io Package
4.2 Java I/O Classes and Interfaces
4.3 File Class
4.4 Stream Classes
4.4.1 Byte Streams
4.4.2 Character Streams
4.5 Serialization

1. ***Swing***
1.1 JFrame
1.2 JLabel
1.3 ImageIcon
1.4 JTextField
1.5 JPasswordField
1.6 Swing Buttons
1.7 JTabbedPane
1.8 JScrollPane

1.9 JList.....
1.10 JComboBox.....
2. ***Event Handling***
2.1 Event Classes.....
2.2 Source of Events.....
2.3 Event Listener Interfaces.....
2.4 Mouse Events.....
2.5 Keyboard Events.....
3. ***JavaFX***
3.1 Introduction to JavaFX.....
3.2 JavaFX Setup.....
3.3 Scene Graph.....
3.3.1 Root.....
3.3.2 Children.....
3.3.3 Hierarchy.....
3.4 Layout Containers.....
3.4.1 VBox.....
3.4.2 HBox.....
3.4.3 GridPane.....
3.4.4 StackPane.....
3.5 Buttons and Controls.....
3.6 Styling in JavaFX.....
3.7 Animation and Transitions in JavaFX.....

1. Introduction to Java

*Java is a **high-level, object-oriented programming language** used for developing a wide variety of software applications. It is designed in such a way that programs written in Java can run on many different types of computers and operating systems without modification.*

*Java is considered a **general-purpose programming language**, which means it can be used to create different types of applications such as desktop software, web applications, mobile applications, enterprise systems, and cloud-based services.*

*Unlike low-level languages that interact directly with hardware, Java provides a **simple and user-friendly programming environment**. It uses English-like syntax and structured programming techniques, which makes it easier for programmers to write, understand, and maintain programs.*

*Java also follows the **object-oriented programming (OOP) paradigm**, where programs are built using objects and classes. This approach helps in organizing large programs into smaller reusable components, making software development more efficient and manageable.*

History and Development of Java

*The development of Java started in **1991** at Sun Microsystems by a group of engineers known as the **Green Team**. The project was led by the famous computer scientist James Gosling, who is often referred to as the **Father of Java**.*

*Initially, the language was designed for **consumer electronic devices**, such as television set-top boxes and home appliances. The main goal was to develop a language that could work on different electronic devices regardless of the hardware platform.*

*The language was originally named **Oak**, after an oak tree that stood outside James Gosling's office. However, since the name Oak was already registered as a trademark, it was later renamed **Java**.*

*Java was officially released to the public in **1995**. During that time, the internet was rapidly expanding, and Java quickly became popular because it allowed developers to create **interactive web applications**.*

*Later in **2010**, Oracle Corporation acquired Sun Microsystems and took responsibility for maintaining and improving the Java platform. Today, Java continues to evolve with regular updates and remains one of the most widely used programming languages.*

Smart Study PLUS

1.1 Features / Characteristics of Java

Java has many important features that make it one of the most popular programming languages in modern software development. These features were designed to make programming ***easy, secure, portable, and efficient****. Because of these advantages, Java is widely used to develop* ***web applications, mobile applications, enterprise systems, and desktop software****.*

The major features or characteristics of Java are explained below in simple language.

1. Simple

Java is considered a ***simple programming language*** *because its syntax is easy to learn and understand. It was designed to remove many complicated features that existed in earlier languages like C and C++.*

For example, Java does not use ***pointers****, which often make programs difficult to manage. It also simplifies concepts such as memory management. Because of this, programmers can focus more on solving problems rather than dealing with complex language features.*

Due to its clear structure and readable syntax, Java is suitable for ***both beginners and professional developers****.*

2. Object-Oriented

Java is based on the concept of ***object-oriented programming (OOP)****. In this approach, programs are created using* ***classes and objects****.*

Objects represent real-world entities, and they contain ***data (variables) and methods (functions)*** *that work on that data. This structure helps organize programs in a better way.*

Java supports the main principles of object-oriented programming, such as:

- ***Encapsulation*** *– combining data and methods into a single unit*
- ***Inheritance*** *– allowing one class to use the properties of another class*
- ***Polymorphism*** *– using one method name with different behaviors*
- ***Abstraction*** *– hiding unnecessary details and showing only essential features*

These principles help developers create ***reusable and well-organized programs****.*

3. Platform Independent

One of the most important features of Java is ***platform independence****.*

In many programming languages, programs must be compiled separately for each operating system. However, Java programs are first converted into ***bytecode****, which can run on any system that has the* ***Java Virtual Machine (JVM)*** *installed.*

This means a program written on one computer can run on another computer without changing the code.

This concept is known as ***"Write Once, Run Anywhere (WORA)"****.*

Because of this feature, Java programs can work on different operating systems such as ***Windows, Linux, and macOS****.*

4. Secure

Security is a very important feature of Java. Java programs run inside a ***protected environment****, which helps prevent harmful operations.*

Java provides several security mechanisms, such as:

- ***Bytecode verification***
- ***Class loader system***
- ***Restricted access to system memory***
- ***Security manager***

These mechanisms ensure that Java programs cannot easily damage the computer system or access unauthorized resources.
For this reason, Java is widely used in ***network-based applications and online services****.*

5. Robust

Java is designed to be a ***strong and reliable programming language****. It includes several features that help detect and handle errors effectively.*
Some of these features include:

- ***Automatic memory management***
- ***Garbage collection***
- ***Exception handling***
- ***Strong data type checking***

These mechanisms help reduce program crashes and make Java applications more stable and dependable.

6. Multithreaded

Java supports ***multithreading****, which means multiple tasks can run at the same time within a single program.*
A ***thread*** *is a small part of a program that performs a specific task. By using multiple threads, Java programs can perform several operations simultaneously.*
For example, in a web browser:

- *One thread downloads data from the internet*
- *Another thread displays images or text*
- *Another thread processes user input*

Multithreading improves ***performance and responsiveness*** *of applications.*

7. Distributed

Java provides strong support for ***distributed applications****, which run on multiple computers connected through a network.*
Java includes libraries that help programs communicate over networks, such as:

- ***Remote Method Invocation (RMI)***
- ***Networking packages***
- ***Web service technologies***

These features make Java suitable for ***client–server systems and internet-based applications****.*

8. Portable

Java programs are ***portable****, meaning they can easily run on different systems without major changes.*

*This is possible because Java uses **standard data types and platform-independent bytecode**.*
As a result, the behavior of Java programs remains consistent on different machines.
Portability allows developers to develop software once and run it on many platforms.

9. High Performance

*Although Java programs run through the **Java Virtual Machine (JVM)**, they still achieve good performance.*
*This is because Java uses **Just-In-Time (JIT) compilation**, which converts bytecode into machine code during program execution. This process increases the speed of the program.*
*As a result, Java can efficiently handle **large and complex applications**.*

10. Dynamic

*Java is also considered a **dynamic programming language** because it can adapt to changes during runtime.*
Java supports features such as:

- ***Dynamic memory allocation***
- ***Dynamic class loading***
- ***Runtime method binding***

These features allow Java programs to be more flexible and adaptable during execution.

1.2 Java Bytecode Architecture

*Java follows a unique program execution model known as **Java Bytecode Architecture**. This architecture explains how a Java program is converted from **human-readable source code into a machine-executable form** while still maintaining **platform independence**.*
*In most programming languages, the compiler directly converts the program into **machine code** that works only on a specific computer or operating system. However, Java follows a **two-step compilation and execution process**. First, the program is converted into **bytecode**, and then the bytecode is executed by the **Java Virtual Machine (JVM)**.*

*Because of this architecture, Java programs are **not dependent on any particular hardware or operating system**. This design allows Java programs to run on different systems without modification.*

Basic Concept of Java Bytecode Architecture

*The main idea behind Java Bytecode Architecture is to **create an intermediate layer between the program and the computer system**.*

*Instead of interacting directly with the hardware, the Java program interacts with the **Java Virtual Machine**. The JVM then translates the program instructions into machine code that the system can execute.*

*This process can be understood as a **three-level execution system**:*

1. ***Java Source Code***

2. ***Java Bytecode***
3. ***Machine Code through JVM***

By introducing bytecode between source code and machine code, Java ensures that the program can be executed on any system that has a compatible JVM.

Step 1: Writing the Java Source Program

*The first step in the Java Bytecode Architecture is the creation of the **Java source program**.*
A programmer writes the program using the Java programming language. The program is written in a text file using Java syntax and is saved with the extension:
.java
This source code contains program logic such as:

- *Class definitions*
- *Variables*
- *Methods*
- *Control statements*
- *Object creation*

Although the source code is understandable to humans, it cannot be executed directly by the computer.

Step 2: Compilation into Bytecode

*The next step is the **compilation process**.*
*Java provides a compiler called **javac**. This compiler reads the Java source program and checks for **syntax errors**. If the program is written correctly, the compiler converts the source code into **Java Bytecode**.*
The output of this compilation process is stored in a file with the extension:
.class
Each class in the Java program generates its own .class file.
*The important concept here is that the compiler does **not generate machine code**. Instead, it generates **bytecode**, which is a platform-neutral instruction set.*

Step 3: Bytecode as an Intermediate Representation

*Java Bytecode acts as an **intermediate representation of the program**.*
It is a compact set of instructions that represent the operations of the Java program. These instructions include operations such as:

- *Loading values into memory*
- *Performing arithmetic calculations*
- *Calling methods*
- *Controlling program flow*

*Bytecode instructions are designed specifically for the **Java Virtual Machine**.*
*Because bytecode is **not tied to any particular processor or operating system**, it can be executed on different platforms.*
*This intermediate layer is the key reason why Java supports **cross-platform execution**.*

Step 4: Role of the Java Virtual Machine (JVM)

*The **Java Virtual Machine (JVM)** is the core component responsible for executing Java bytecode.*

*The JVM acts as a **software-based machine** that understands bytecode instructions. It performs several important functions during program execution.*

*First, the JVM **loads the bytecode into memory**. Then it verifies the bytecode to ensure that it follows Java's security rules and does not perform illegal operations.*

*After verification, the JVM converts the bytecode into **machine instructions** that the computer's processor can execute.*

This conversion may happen in two ways:

- ***Interpretation**, where the JVM reads and executes bytecode instructions one by one*
- ***Just-In-Time (JIT) compilation**, where bytecode is converted into machine code at runtime to improve performance*

Because each operating system has its own JVM implementation, the same Java bytecode can run on different systems.

Step 5: Execution on Operating System and Hardware

*Once the JVM converts bytecode into machine instructions, the program is executed by the **computer's processor through the operating system**.*

At this stage, the program interacts with system resources such as:

- *Memory*
- *Input and output devices*
- *Files*
- *Network connections*

*Even though the program eventually runs on the hardware, the Java program itself **never directly interacts with the hardware**. Instead, it communicates with the JVM, which acts as an intermediary.*

*This layered approach ensures that Java programs remain **portable and secure**.*

Java Bytecode Architecture

The entire architecture can be understood through the following conceptual flow:

Java Source Program (.java)

↓

Java Compiler (javac)

↓

Bytecode (.class)

↓

Java Virtual Machine

↓

Machine Code Execution

↓

Program Output

*This structure clearly shows how Java separates **program logic from system hardware**.*

Advantages of Java Bytecode Architecture

The Java Bytecode Architecture provides several important benefits.
*One major advantage is **platform independence**. Because bytecode is executed by the JVM, the same program can run on different systems without modification.*
*Another advantage is **security**. The JVM verifies bytecode before execution, preventing harmful programs from damaging the system.*
*It also improves **portability**, allowing Java programs to be easily transferred between different machines.*
*The architecture also supports **efficient execution** through technologies such as Just-In-Time compilation.*
*Finally, this architecture enables **network-based program execution**, allowing Java programs to be downloaded and executed across the internet.*

1.3 Java Program Structure

*Java programs are written in a **well-defined structured format**. This structure organizes the different parts of the program in a logical way so that the compiler, the Java Virtual Machine (JVM), and programmers can easily understand and execute the program.*
*A Java program is not just a set of instructions written randomly. Instead, it is composed of several **organized components**, each performing a specific role in the program. These components help in **code organization, modular programming, readability, and program execution**.*

A typical Java program structure contains the following components:

1. *Package Declaration*
2. *Import Statements*
3. *Comments*
4. *Class Definition*
5. *Variables (Class variables and Local variables)*
6. *Methods / Behaviors*
7. *Main Method*

Each of these components is explained in detail below.

1. Package Declaration

*The **package declaration** is used to specify the package to which the Java class belongs. A package is a collection of related classes and interfaces grouped together for better organization.*

In large software applications, thousands of classes may exist. Without packages, it would become difficult to manage these classes. Packages help organize classes into logical groups and prevent ***name conflicts*** *between classes with the same name.*
Conceptually, a package works like a ***folder or directory*** *in a computer system. Just as files are stored in folders, Java classes are stored in packages.*
For example, Java provides many built-in packages such as:

- *java.lang*
- *java.util*
- *java.io*
- *java.net*

The package declaration must appear ***at the beginning of the program****, before any class or interface definition.*
Syntax
package package_name;
Example
package studentmanagement;
This statement indicates that the class belongs to the ***studentmanagement package****.*

2. Import Statements
The ***import statement*** *allows a Java program to access classes and interfaces that are defined in other packages.*
Java provides a large number of ***predefined libraries*** *that contain useful classes. Instead of writing everything from scratch, programmers can reuse these classes by importing them.*
Conceptually, the import statement acts as a ***bridge between the current program and external classes****.*
There are two ways to import classes:
Importing an Entire Package
This allows the program to use all classes from a package.
import java.util.;*
Importing a Specific Class
This allows the program to use only a specific class from a package.
import java.util.Scanner;
Using import statements improves ***code reusability and modularity****, which are important principles of software development.*

3. Comments
Comments are used to provide ***explanations and descriptions within the source code****. They help programmers understand the purpose and functionality of different parts of the program.*
Comments are ***non-executable statements****, which means the compiler ignores them during program execution.*
Conceptually, comments serve as ***documentation inside the program****.*
They are very useful when:

- *Explaining complex logic*
- *Providing information about variables or methods*
- *Temporarily disabling code during debugging*

Java supports three types of comments.

Single-Line Comment

Used for short explanations.

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

Multi-Line Comment

Used for longer descriptions.

```
/* This is a
   multi-line comment */
```

Documentation Comment

Used for generating program documentation.

```
/**
 * This program explains Java structure
 */
```

*Good commenting practice improves **program readability and maintainability**.*

4. Class Definition

*The **class definition** is the central part of a Java program. Every Java program must contain at least **one class** because Java is a **pure object-oriented programming language**.*

*A class is a blueprint or template used to create objects. It defines the **properties (variables)** and **behaviors (methods)** of objects.*

*Conceptually, a class represents a **real-world entity**.*

For example:

- *Student*
- *Employee*
- *Car*
- *BankAccount*

A class contains:

- *Variables (data members)*
- *Methods (functions)*
- *Constructors*

Syntax

```
class ClassName
{
   // variables
   // methods
}
```

Example

```
class Student
{
}
```

*A Java program may contain multiple classes, but only one class can be declared as **public**.*

5. Variables

*Variables are used to **store data values during program execution**. These values may change as the program runs.*

*Conceptually, a variable represents a **memory location that stores a specific type of data**.*

*In Java, variables must be declared with a **data type**, which defines the type of value that can be stored.*

Syntax

data_type variable_name;

Example:

int age;

String name;

Java mainly uses two types of variables.

Class Variables (Instance Variables)

These variables are declared inside the class but outside methods.

*They represent the **properties of an object** and can be accessed by all methods of the class.*

Example:

int rollNumber;

Local Variables

Local variables are declared inside methods or blocks.

They exist only within the scope of the method in which they are declared.

Example:

int sum = 10;

*Variables help programs **store, process, and manipulate data** during execution.*

6. Methods / Behaviors

*A **method** is a block of code that performs a specific task. It defines the behavior of a class.*

*Conceptually, methods represent **actions performed by objects**.*

*For example, if a class represents a **Student**, possible methods might be:*

- *displayDetails()*
- *calculateMarks()*
- *registerCourse()*

*Methods help in **breaking large programs into smaller manageable units**.*

This improves:

- *Code readability*
- *Reusability*
- *Maintainability*

Syntax

returnType methodName(parameters)

{

```
    // statements
}
```

Example:

```
void displayMessage()
{
    System.out.println("Welcome to Java");
}
```

Methods can also accept parameters and return values.

7. Main Method

*The **main() method** is the starting point of program execution in Java.*
*When a Java program runs, the **Java Virtual Machine (JVM)** first looks for the main method. Execution begins from this method.*
Without the main method, a Java program cannot run.
*Conceptually, the main method acts as the **entry point of the program**.*

Syntax

```
public static void main(String args[])
{
    // program statements
}
```

Explanation of components:

public
The method is accessible to the JVM from anywhere.
static
Allows the method to run without creating an object of the class.
void
Indicates that the method does not return any value.
main
Name of the method recognized by the JVM.
String args[]
Used to accept command-line arguments.

Execution of Java Program (Step-by-Step Process)

*Java follows a special execution process to run programs on different systems. Unlike many other programming languages, Java does not directly convert source code into machine code. Instead, it follows a **multi-stage process** that includes compilation, bytecode generation, and execution through the **Java Virtual Machine (JVM)**.*
*This process ensures that Java programs are **platform independent, secure, and portable**. The internal working of a Java program can be understood through several important stages.*

1. Writing the Java Source Program

*The first step in the execution of a Java program is **writing the source code**.*
A programmer writes the program using the Java language with the help of a text editor or an Integrated Development Environment (IDE). The program includes elements such as:

- *Classes*
- *Variables*
- *Methods*
- *Statements*
- *Objects*

The source program is saved with the file extension:
.java
For example:
Hello.java
At this stage, the program is understandable to humans but ***cannot be executed directly by the computer****.*

2. Compilation of the Program
After writing the program, the next step is ***compilation****.*
Java provides a compiler called ***javac*** *that converts the Java source code into an intermediate form called* ***bytecode****.*
During compilation, the compiler performs several tasks:

- *Checks the program for* ***syntax errors***
- *Verifies that Java language rules are followed*
- *Converts the source code into bytecode instructions*

If the program contains errors, the compiler displays error messages. The programmer must correct these errors before the program can be compiled successfully.
After successful compilation, the compiler creates a file with the extension:
.class
For example:
Hello.class
This file contains the ***Java bytecode****.*

3. Generation of Bytecode
The compiled .class file contains ***bytecode****, which is a set of instructions designed for the* ***Java Virtual Machine****.*
Bytecode is not machine code and is ***not specific to any operating system or processor****.*
Instead, it is a standardized intermediate representation of the program.

Because of this design, the same bytecode can run on different systems that have a JVM installed.
This is the main reason why Java supports the concept:
"Write Once, Run Anywhere."

4. Class Loading
After compilation, the next step is ***loading the bytecode into memory****.*
This task is performed by a component of the JVM called the ***Class Loader****. The class loader reads the .class file and loads it into the JVM memory when the program starts.*

The class loader performs several operations such as:

- *Locating the required class files*
- *Loading them into memory*
- *Linking them with other classes if necessary*

This process prepares the program for execution.

5. Bytecode Verification

*Before executing the program, the JVM performs a **verification process**.*
*The **Bytecode Verifier** checks whether the bytecode follows Java's safety rules. It ensures that the program does not perform illegal operations such as:*

- *Accessing invalid memory locations*
- *Violating access permissions*
- *Corrupting the system environment*

*This verification process improves the **security and reliability** of Java programs.*

6. Execution by the JVM

Once the bytecode is verified, the JVM begins executing the program.
*The JVM converts the bytecode instructions into **machine instructions** that the computer's processor can understand. This conversion may happen in two ways:*
Interpreter
The JVM reads the bytecode instructions one by one and executes them sequentially.
Just-In-Time (JIT) Compiler
The JIT compiler converts frequently used bytecode into machine code during runtime to improve execution speed.
Both methods help the JVM run the program efficiently.

7. Memory Management and Garbage Collection

While the program is running, the JVM manages memory automatically.
*Java uses a feature called **Garbage Collection**, which automatically removes objects that are no longer used by the program. This helps free memory and prevents memory leaks.*
Because of automatic memory management, programmers do not need to manually allocate or deallocate memory.

8. Program Execution and Output

*After the JVM processes the bytecode and converts it into machine instructions, the program is executed by the **computer's processor**.*
*The program performs the operations defined in the source code and produces the required **output**.*
The output may appear in different forms such as:

- *Text displayed on the screen*
- *Data written to a file*
- *Network communication*

- *Graphical user interface output*

Overall Flow of Java Program Execution

The complete internal working of a Java program can be summarized as follows:

Java Source Program (.java)
↓
Java Compiler (javac)
↓
Bytecode (.class)
↓
Class Loader
↓
Bytecode Verification
↓
JVM Execution Engine
↓
Machine Code
↓
Program Output

3. Token Expressions

3.1 Tokens in Java

Java programs are written using a collection of small elements known as ***tokens****. A token is the* ***smallest meaningful unit of a Java program that the compiler can recognize and process during compilation****.*

When a Java program is compiled, the compiler does not interpret the program as a continuous sequence of sentences. Instead, it performs a process called ***lexical analysis****, in which the source code is divided into individual units called tokens. Each token represents a specific component of the programming language such as keywords, identifiers, constants, operators, or separators.*

Tokens play a crucial role in helping the compiler understand the ***syntax, structure, and meaning of the program****. They form the* ***basic building blocks of a Java program****, and every statement written in Java consists of one or more tokens.*

Example

int sum = a + b;
The tokens in this statement are:
int , sum , = , a , + , b , ;
Each token represents a specific part of the instruction.
int → keyword
sum → identifier
= → assignment operator
a → identifier
+ → arithmetic operator

b → identifier
; → separator
Thus, tokens allow the compiler to analyze the program systematically.

Categories of Tokens in Java
Tokens in Java are classified into the following categories:

1. *Keywords*
2. *Identifiers*
3. *Constants (Literals)*
4. *Operators*
5. *Separators (Punctuators)*
6. *Special Symbols*

Each category performs a specific function in the structure of a Java program.

1. Keywords
*Keywords are **reserved words that have predefined meanings in the Java programming language**. These words are part of the language syntax and are used by the compiler to perform specific tasks such as defining classes, declaring variables, controlling program flow, and specifying access modifiers.*
*Since keywords have fixed meanings, they **cannot be used as identifiers** such as variable names, class names, or method names.*

Java contains a large number of keywords that are used in different programming constructs.

Examples of Java Keywords
class int float public private protected
static void if else while for
switch return new this super
Example

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

In this example:

- *class defines the creation of a class.*
- *int specifies the integer data type.*

*Keywords help define the **structure and control mechanisms of a Java program**.*

2. Identifiers
*Identifiers are **user-defined names given to various program elements** such as variables, methods, classes, objects, packages, and interfaces. They serve as symbolic names that allow programmers to reference these elements within the program.*
*Identifiers play an important role in improving the **clarity, readability, and maintainability** of a program.*

Example
int totalMarks;
Here, totalMarks is an identifier.

Rules for Naming Identifiers
Java identifiers must follow specific rules:

1. *An identifier must start with a **letter, underscore (_), or dollar sign ($)**.*
2. *It **cannot begin with a digit**.*
3. *Digits may appear **after the first character**.*
4. *Identifiers **cannot contain spaces**.*
5. *Special characters such as @, #, %, &, *, ! are not allowed.*
6. *Keywords cannot be used as identifiers.*
7. *Identifiers are **case-sensitive**, meaning value and Value are different.*
8. *Identifiers should be **meaningful and descriptive**.*

Valid Identifiers	***Invalid Identifiers***
sum	*1value*
studentAge	*total marks*
_total	*class*
price1	*@data*

*Using clear and meaningful identifiers improves **program readability and program design**.*

3. Constants (Literals)
*Constants, also called **literals**, represent fixed values that do not change during program execution. These values are written directly in the program code.*
Constants allow programmers to represent specific data values clearly and reduce errors during execution.

Types of Constants in Java

Type of Constant	***Description***	***Examples***
Integer Constants	*Represent whole numbers without any decimal or fractional part.*	*10, -45, 0, 1000*
Floating-Point Constants	*Represent decimal numbers or numbers with fractional parts.*	*3.14, -0.05, 2.5e3*
Character Constants	*Represent a single character enclosed within single quotation marks.*	*'A', '9', '\n'*
String Constants	*Represent a sequence of characters enclosed within double quotation marks.*	*"Hello", "Java Programming"*
Boolean	*Represent logical values used in decision-*	*true, false*

Constants	*making statements.*	

4. Operators

*Java uses **operators** to perform different types of operations on data values and variables. Operators are special symbols that instruct the compiler to perform specific mathematical, relational, logical, or assignment operations.*

*In programming, data stored in variables often needs to be **processed, compared, or modified**. Operators make it possible to manipulate data and control the behavior of the program. Every operator works with one or more **operands** (values or variables on which the operation is performed).*

For example:

int sum = a + b;

Here:

- *+ is the **operator***
- *a and b are **operands***

*Operators are essential because they allow programs to **perform calculations, make decisions, and update variable values during execution**.*

Major Types of Operators in Java

Java provides several categories of operators. The most commonly used ones include:

1. *Arithmetic Operators*
2. *Relational Operators*
3. *Logical Operators*
4. *Assignment Operators*
5. *Increment and Decrement Operators*

These operators are explained below in detail.

1. Arithmetic Operators

*Arithmetic operators are used to perform **mathematical calculations** on numeric values.*

They are commonly used in programs that involve calculations such as addition, subtraction, multiplication, and division.

*These operators work with **numeric data types** such as int, float, double, and long.*

Arithmetic Operators Table

Operator	***Operation***	***Description***	***Example***
+	*Addition*	*Adds two operands*	*a + b*
-	*Subtraction*	*Subtracts second operand from first*	*a - b*
*	*Multiplication*	*Multiplies two operands*	*a * b*

/	Division	Divides one operand by another	a / b
%	Modulus	Returns remainder after division	a % b

Example

```
int a = 10;
int b = 3;
System.out.println(a + b); // 13
System.out.println(a % b); // 1
```

*Arithmetic operators are fundamental for **performing numerical computations**.*

2. Relational Operators

*Relational operators are used to **compare two values**. The result of a relational operation is always a **Boolean value** (true or false).*
*These operators are commonly used in **conditional statements and loops**.*

Relational Operators Table

Operator	Description	Example	Result
==	Equal to	a == b	true if values are equal
!=	Not equal to	a != b	true if values are different
>	Greater than	a > b	true if a is greater
<	Less than	a < b	true if a is smaller
>=	Greater than or equal to	a >= b	true if condition satisfied
<=	Less than or equal to	a <= b	true if condition satisfied

Example

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

System.out.println(a > b);  // true
System.out.println(a == b); // false
```

*Relational operators help programs **make decisions based on comparisons**.*

3. Logical Operators

*Logical operators are used to **combine multiple conditions or expressions**. They are mainly used in **decision-making statements such as if, while, and for loops**.*
*Logical operators operate on **Boolean values**.*

Logical Operators Table

Operator	Name	Description	Example
&&	Logical AND	Returns true if both conditions are true	(a > b) && (a > c)
`		`	Logical OR
!	Logical NOT	Reverses the logical value	!(a > b)

Example

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

System.out.println((a > b) && (b > 2)); // true
System.out.println((a < b) || (b > 2)); // true
```

Logical operators allow programs to **evaluate multiple conditions simultaneously**.

4. Assignment Operators

Assignment operators are used to **assign values to variables**. The basic assignment operator assigns the value on the right side to the variable on the left side.

Java also provides **compound assignment operators** that combine arithmetic operations with assignment.

Assignment Operators Table

Operator	Description	Example
=	Assign value	a = 10
+=	Add and assign	a += 5
-=	Subtract and assign	a -= 5
*=	Multiply and assign	a *= 5
/=	Divide and assign	a /= 5
%=	Modulus and assign	a %= 5

Example

```
int a = 10;

a += 5;   // a = a + 5
System.out.println(a); // 15
```

Assignment operators simplify code and **reduce the need for repeated expressions**.

5. Increment and Decrement Operators

Increment and decrement operators are used to **increase or decrease the value of a variable by one**.

They are commonly used in ***loops and counters****.*

Increment and Decrement Operators Table

Operator	***Name***	***Description***	***Example***
++	*Increment*	*Increases value by 1*	*a++*
--	*Decrement*	*Decreases value by 1*	*a--*

Types of Increment

Pre-Increment

Value increases ***before*** *it is used.*

++a

Post-Increment

Value increases ***after*** *it is used.*

a++

Example

```
int a = 5;
a++;
System.out.println(a); // 6
a--;
System.out.println(a); // 5
```

These operators are extremely useful for ***iteration and counting operations****.*

5. Separators (Punctuators)

Separators are special characters used to ***separate statements and organize program structure****. They help define blocks of code and group expressions.*

Common separators in Java include:

; ( ) { } [ ] , .

Examples

; → ends a statement

{ } → defines a block of code

( ) → used in method calls

[ ] → used in arrays

, → separates parameters

Separators ensure ***correct syntax and program organization****.*

6. Special Symbols

Special symbols are characters that have ***specific functional meanings in Java programming****.*

These symbols help perform advanced programming tasks such as member access, annotations, and referencing objects.

Examples include:

. → member access operator
@ → annotation symbol
:: → method reference
$ → used in identifiers

*These symbols enhance Java's **expressiveness and functionality**.*

Example Program Showing Java Tokens

```
import java.util.*;        // import → keyword

public class TokenDemo     // public, class → keywords
{
   public static void main(String args[])
   {
      int a = 10, b = 20;      // identifiers, constants, operator
      int sum = a + b;         // arithmetic operator
      System.out.println("Sum = " + sum);  // string constant
   }
}
```

Tokens used in this program include:

- *Keywords → import, public, class, int*
- *Identifiers → TokenDemo, a, b, sum*
- *Constants → 10, 20, "Sum ="*
- *Operators → =, +*
- *Separators → ;, {}, ()*

2.2 Data Types in Java

*Java uses **data types** to specify the nature of the data that can be stored and manipulated within a program. A data type defines the **kind of value a variable can hold, the amount of memory allocated to store that value, and the operations that can be performed on it**.*
*In computer programming, all information is stored in **memory locations**. When a variable is declared, the compiler must know the type of data that will be stored in that variable so that it can allocate the correct amount of memory and apply the correct operations.*

For example:

```
int number = 25;
```

In this statement:

- *int specifies that the variable will store **integer values***
- *number is the **identifier***
- *25 is the **value assigned to the variable***

*The data type int tells the Java compiler to allocate **4 bytes of memory** for storing the integer value.*

Thus, data types act as ***instructions for the compiler on how to store and interpret data in memory****.*

Data Representation in Java

In Java, all data stored in memory is ultimately represented in ***binary form (0s and 1s)****. Data types help interpret these binary values correctly.*

For example:

- *An* ***integer*** *represents a whole number*
- *A* ***floating-point*** *value represents a decimal number*
- *A* ***character*** *represents a symbol encoded using Unicode*
- *A* ***boolean*** *represents a logical state (true or false)*

Without data types, the computer would not know how to interpret the stored binary values.

Need for Data Types

Data types are essential in programming for several reasons:

1. ***Memory Allocation***
   Data types determine how many bytes of memory are required to store a variable.
2. ***Data Integrity***
   They ensure that variables store only appropriate values.
3. ***Efficient Execution***
   Proper data types improve the efficiency of the program.
4. ***Type Safety***
   Java prevents assigning incompatible values to variables.
5. ***Error Detection***
   Data type checking allows the compiler to detect errors during compilation.

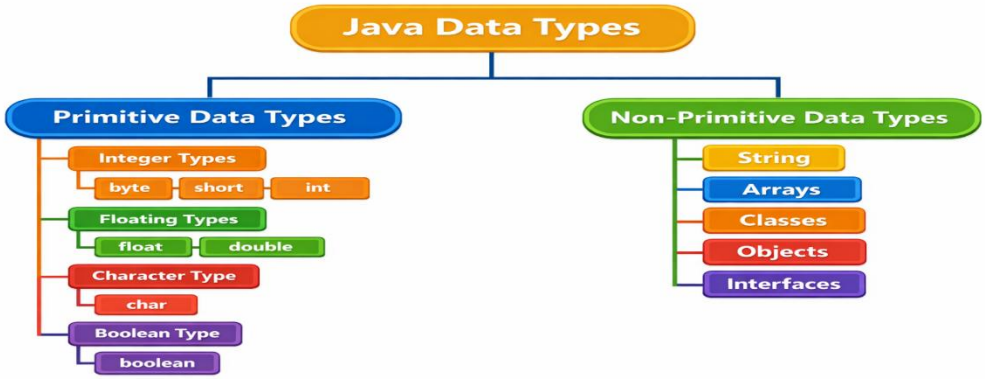
For example:

```
int number = "Hello";   // Error
```

The compiler produces an error because a ***string cannot be stored in an integer variable****.*

Classification of Data Types in Java

Java data types are broadly divided into two categories:



1. Primitive Data Types

*Primitive data types are the **fundamental data types provided directly by the Java language**. They represent **single values and are stored directly in memory**. Java provides **eight primitive data types**.*

Data Type	***Size***	***Default Value***	***Description***
byte	*1 byte*	*0*	*Stores small integer values*
short	*2 bytes*	*0*	*Stores short integers*
int	*4 bytes*	*0*	*Most commonly used integer type*
long	*8 bytes*	*0L*	*Used for very large integers*
float	*4 bytes*	*0.0f*	*Stores decimal numbers*
double	*8 bytes*	*0.0d*	*Stores large decimal values*
char	*2 bytes*	*'\u0000'*	*Stores a single Unicode character*
boolean	*1 bit*	*false*	*Stores logical values*

*Primitive types are efficient because they **directly store the value rather than the reference to the value**.*

Further Classification of Primitive Data Types

Primitive data types are grouped into four major categories.

1. Integer Data Types

*In Java, **integer data types** are used to store **whole numbers without any decimal or fractional part**. These data types represent numeric values such as positive numbers, negative numbers, and zero. Integer data types are commonly used in programs for counting, indexing, storing numerical values, and performing arithmetic operations. Java provides four different integer data types—byte, short, int, and long. Each type differs in the **amount of memory it occupies and the range of values it can store**. Smaller types such as byte and short are used when memory efficiency is important, while int and long are used for storing larger numbers. Among these, int is the most commonly used integer data type in Java programs.*

Type	***Size***	***Range***
byte	*1 byte*	*-128 to 127*
short	*2 bytes*	*-32,768 to 32,767*
int	*4 bytes*	*-2^{31} to $2^{31}-1$*
long	*8 bytes*	*-2^{63} to $2^{63}-1$*

Example:

int marks = 95;
long population = 8000000000L;

2. Floating-Point Data Types

*In Java, **floating-point data types** are used to store **numbers that contain decimal or fractional values**. These types are useful when a program needs to represent values that require precision, such as measurements, scientific calculations, or financial data. Java provides two floating-point data types: float and double. The float data type occupies **4 bytes of memory** and is used for storing decimal numbers with moderate precision, while the double data type occupies **8 bytes of memory** and provides higher precision for storing larger decimal values. Among these, double is more commonly used in Java programs because it offers greater accuracy when performing calculations involving decimal numbers.*

Type	***Size***	***Precision***
float	*4 bytes*	*~7 decimal digits*
double	*8 bytes*	*~15 decimal digits*

Example:

float price = 12.5f;
double distance = 12345.6789;

*The double type provides **higher precision** than float.*

4. Character Data Type

The character data type (char) in Java is used to store a single character such as a letter, digit, or special symbol. A character value is always enclosed within single quotation marks. The char data type occupies 2 bytes of memory because Java uses the Unicode character encoding system, which allows it to represent characters from many languages around the world. Characters such as 'A', '7', or '@' can be stored using this data type. The char data type is commonly used in programs when dealing with individual characters, symbols, or text processing tasks.
*The char data type stores **a single character** such as a letter, digit, or symbol.*

Example:

char grade = 'A';
*Java uses **Unicode encoding**, which supports characters from multiple languages.*
For example:
char symbol = '@';

4. Boolean Data Type

*The **boolean data type** in Java is used to represent **logical values** in a program. It can store only two possible values: **true** or **false**. This data type is mainly used in situations where a program needs to make decisions based on certain conditions. For example, boolean values are commonly used in conditional statements such as if, while, and for to control the flow of*

program execution. The boolean data type does not store numbers or characters; it only represents logical states. By evaluating conditions that result in either true or false, the boolean data type helps programs perform ***decision-making and logical operations effectively****.It can store only two values:*

- *true*
- *false*

Example:

boolean isEligible = true;
Boolean values are widely used in ***decision-making and conditional statements****.*

Non-Primitive (Reference) Data Types :
In Java, ***non-primitive data types*** *are used to store* ***references (addresses) to objects rather than the actual values themselves****. These data types are more complex than primitive types and are essential for implementing* ***object-oriented programming concepts****.*
Non-primitive data types allow programmers to create ***structured data, reusable code components, and real-world models*** *within programs. Unlike primitive types, which store a single value, reference types can represent* ***collections of values, objects, or behaviors****.*

The most common non-primitive data types in Java are:

- *String*
- *Arrays*
- *Classes*
- *Objects*
- *Interfaces*

Each of these is explained in detail below.

1. String
In Java, a ***String*** *is a reference data type used to store a* ***sequence of characters****, such as words, sentences, or text. Strings are written inside* ***double quotation marks****. In Java, a string is actually an* ***object of the String class****, which provides many built-in methods for handling and manipulating text. Strings are commonly used to store information such as* ***names, messages, and descriptions*** *in a program.*
A string is ***an object that stores a sequence of characters in memory****. In Java, strings are instances of the String class provided in the java.lang package.*

Unlike primitive data types, a string stores a ***reference to an object that contains the character data****.*

Example
String name = "Rahul";
In this example:

- *String → reference data type*
- *"Rahul" → string object stored in memory*
- *name → reference variable pointing to the object*

Important Characteristics of Strings

- *Strings are **immutable**, meaning their values cannot be changed once created.*
- *If a string is modified, Java actually creates **a new string object**.*
- *Strings provide many built-in methods such as:*
    - *length()*
    - *toUpperCase()*
    - *substring()*
    - *equals()*

Example

String message = "Hello";
System.out.println(message.length());
*Strings are widely used in programs to handle **text processing and user input**.*

2. Arrays

*In Java, an **array** is a reference data type used to store **multiple values of the same data type in a single variable**. Instead of creating separate variables for each value, an array allows all values to be stored together and accessed using an **index number**. The index of an array starts from **0**, which represents the first element. Arrays are useful when a program needs to manage a **collection of similar data items**, such as marks of students, list of numbers, or names. Arrays are reference types because the variable stores the **memory address of the array object** rather than the values themselves.*

Example

int numbers[] = {10, 20, 30, 40};
In this example:

- *numbers is a reference variable*
- *The array object contains multiple integer values*

Characteristics of Arrays

- *Arrays store **multiple elements of the same type**.*
- *The size of an array is **fixed after creation**.*
- *Elements are accessed using **index numbers starting from 0**.*

Example:

System.out.println(numbers[0]);
*Arrays are commonly used in programs that require **data collections such as lists, records, or tables**.*

3. Classes

*In Java, a **class** is a blueprint or template used to create objects. It defines the **properties (variables) and behaviors (methods)** that the objects of the class will have. A class helps organize data and functions together in a single unit, which is an important concept of **object-oriented programming**. In simple terms, a class describes **what an object is and what it can do**, while the actual objects are created from that class.*
*a class represents **a real-world entity or concept**.*
For example:

- *Student*

- *Car*
- *BankAccount*

A class contains:

- *Variables (data members)*
- *Methods (functions)*

Example

```
class Student
{
   int rollNo;
   String name;
}
```

Here:

- *Student is a class*
- *rollNo and name are variables representing student properties*

*Classes allow programmers to create **structured and reusable program components**.*

4. Objects

*In Java, an **object** is an **instance of a class**. It represents a real-world entity and contains **data (variables) and behavior (methods)** defined by the class. Objects are created using the new keyword and allow a program to use the properties and functions of the class. In simple terms, a class is a **blueprint**, and an object is the **actual item created from that blueprint**. an object contains:*

- ***State** → values stored in variables*
- ***Behavior** → actions performed by methods*
- ***Identity** → unique reference in memory*

Example

```
Student s1 = new Student();
```

Here:

- *Student → class*
- *s1 → object*
- *new → keyword used to create the object*

The object stores the actual data defined in the class.

Example:

```
s1.rollNo = 101;
s1.name = "Anil";
```

*Objects enable Java programs to represent **real-world concepts and relationships**.*

5. Interfaces

*Java, an **interface** is a reference data type used to define **method declarations that a class must implement**. It acts like a **contract that specifies what a class should do**, but it does not contain the full implementation of the methods. Interfaces mainly contain **abstract methods and constants**, and they cannot be used to create objects directly. They help achieve **abstraction and multiple inheritance** in Java. **Example***

```
interface Animal
{
   void sound();
}
```

Any class implementing this interface must provide the sound() method.

Example:

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

2.3 Variables in Java

*In Java, a **variable** is a named memory location used to store data that can change during the execution of a program. Variables allow programmers to store values such as numbers, characters, or objects and use them later in the program. Each variable must be declared with a **data type**, which determines the type of value that can be stored and the amount of memory allocated for it.*

For example:

```
int number = 10;
```

In this statement:

- *int is the **data type***
- *number is the **variable name***
- *10 is the **value stored in the variable***

*Variables are essential because they help programs **store, process, and manipulate data during execution**.*

Types of Variables in Java

Java variables are generally classified into the following types:

1. ***Reference Variables***
2. ***Local Variables***
3. ***Static Variables***
4. ***Final Variables***

Each type has a specific purpose and scope in a Java program.

1. Reference Variables in Java

*In Java, a **reference variable** is a variable that stores the **memory address (reference) of an object instead of storing the actual data value**. These variables are mainly used with **non-primitive data types**, such as classes, arrays, strings, and interfaces.*

*Unlike primitive variables that store values directly in memory, reference variables store the **location of the object in memory**, which allows the program to access and manipulate the object's data and methods.*

Reference Variables

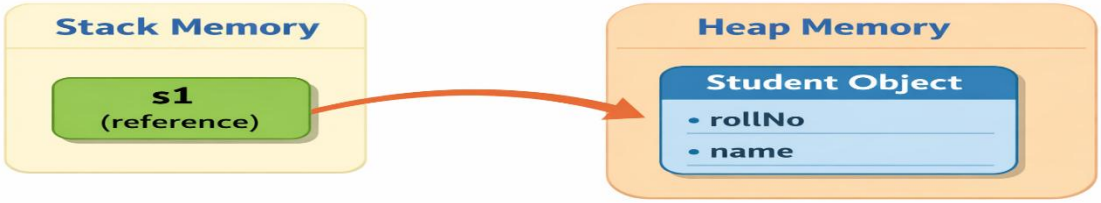
When an object is created in Java using the new keyword, the object is stored in a special memory area called the ***heap memory****. The reference variable then stores the* ***address of that object****, allowing the program to interact with it.*

This means the reference variable acts like a ***pointer or link to the object stored in memory****.*

Example:
Student s1 = new Student();
Explanation:

- *Student → class name*
- *new Student() → creates a new object in memory*
- *s1 → reference variable that stores the memory address of the object*

Thus, s1 does not contain the actual object but ***points to the object stored in memory****.*
Memory Representation



Here:

- *The reference variable s1 is stored in* ***stack memory****.*
- *The actual object created using new is stored in* ***heap memory****.*
- *The reference variable points to the object's memory location.*

Characteristics of Reference Variables
Reference variables have several important characteristics:

1. *They store* ***references (addresses) of objects****, not actual data values.*
2. *They are used with* ***non-primitive data types****.*
3. *Multiple reference variables can point to the* ***same object in memory****.*
4. *When no reference variable points to an object, the object becomes* ***eligible for garbage collection****.*
5. *Reference variables allow objects to* ***access their variables and methods****.*

Example Program

```
class Student
{
   int rollNo;
   String name;
}

public class Example
{
   public static void main(String args[])
   {
      Student s1 = new Student();  // reference variable

      s1.rollNo = 101;
      s1.name = "Rahul";

      System.out.println(s1.rollNo);
      System.out.println(s1.name);
   }
}
```

In this example:

- *s1 is a **reference variable***
- *It refers to the object created from the Student class.*

Multiple Reference Variables

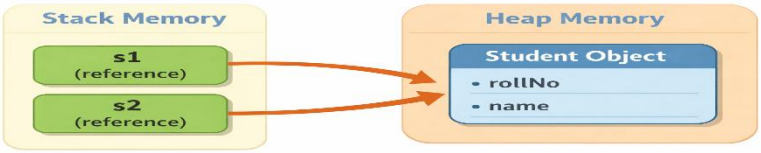
Sometimes, more than one reference variable can refer to the same object.

Example:

```
Student s1 = new Student();
Student s2 = s1;
```

Here:

- *s1 and s2 both refer to the **same object in memory**.*



Any change made using s1 will also affect s2 because both reference variables point to the same object.

2. Local Variables in Java

*In Java, a **local variable** is a variable that is declared **inside a method, constructor, or block of code** and is used only within that specific part of the program. Local variables are temporary variables that store data required during the execution of a particular method or block.*

*Conceptually, a local variable represents **temporary storage created during the execution of a method**. When the method starts executing, memory is allocated for the local variables. Once the method execution finishes, the memory used by these variables is automatically released. Because of this behavior, local variables exist only for a **short period of time during program execution**.*

*Local variables are stored in **stack memory**, which is used for method execution and temporary data storage. Each time a method is called, a new stack frame is created and the local variables are stored in that frame.*

Scope of Local Variables

*The **scope** of a local variable refers to the region of the program where the variable can be accessed.*

A local variable can be used only within:

- *the method in which it is declared*
- *the constructor where it is defined*
- *the block of code where it is created*

Once the execution leaves that block or method, the local variable becomes inaccessible.

Example:

```
public class Example
{
   void display()
   {
      int number = 10;   // local variable
      System.out.println(number);
   }
}
```

Here:

- *number is a local variable*
- *it can only be used inside the display() method*

Trying to access it outside the method will produce a compilation error.

Life Cycle of a Local Variable

Local variables follow a specific life cycle during program execution.

1. ***Declaration** – The variable is declared inside a method or block.*
2. ***Memory Allocation** – Memory is allocated when the method starts executing.*
3. ***Usage** – The variable is used during method execution.*
4. ***Destruction** – Memory is released when the method execution ends.*

*Thus, local variables exist **only during the execution of the method**.*

Characteristics of Local Variables

Local variables have several important properties:

1. *They are declared **inside methods, constructors, or blocks**.*
2. *Their **scope is limited** to the block in which they are declared.*
3. *They are stored in **stack memory**.*
4. *They **do not have default values**.*
5. *They **must be initialized before use**.*
6. *They are **created when the method starts and destroyed when it ends**.*

Example Program

```
public class LocalVariableDemo
{
   public static void main(String args[])
   {
      int a = 10;   // local variable
      int b = 20;   // local variable
      int sum;

      sum = a + b;

      System.out.println("Sum = " + sum);
   }
}
```

In this program:

- *a, b, and sum are **local variables** declared inside the main() method.*
- *They exist only during the execution of the main() method.*

3. Static Variables in Java

*In Java, a **static variable** is a variable that belongs to the **class rather than to any individual object**. It is declared using the keyword static. Unlike instance variables, which are created separately for each object, only **one copy of a static variable exists in memory**, and this single copy is **shared by all objects of the class**.*

*Conceptually, a static variable represents **common data that is shared among all objects of a class**. Since the variable belongs to the class itself, it can be accessed without creating an object of that class. Static variables are commonly used when a particular value or property must be **the same for all objects**, such as a college name for students or a company name for employees.*

*Static variables are stored in a special memory area called the **method area (or class area)** of memory, which is part of the Java runtime environment.*

Syntax

static dataType variableName;

Example:

static int count;

Example Program

```
class Student
{
   int rollNo;
   String name;
   static String college = "ABC College";
}

public class Test
{
   public static void main(String args[])
   {
      Student s1 = new Student();
      Student s2 = new Student();

      System.out.println(s1.college);
      System.out.println(s2.college);
   }
}
```

In this example:

- *college is a **static variable**.*
- *Both s1 and s2 share the same value "ABC College".*

Characteristics of Static Variables

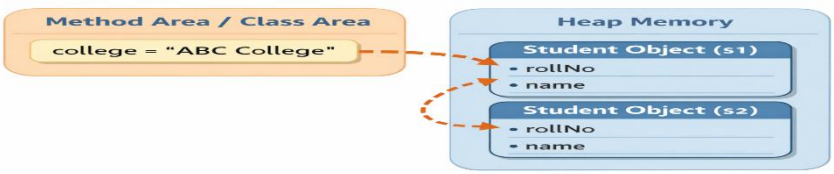
Static variables have the following characteristics:

- *Declared using the **static keyword**.*
- *Belong to the **class rather than objects**.*
- *Only **one copy exists in memory**.*
- *Shared by **all objects of the class**.*
- *Can be accessed using the **class name**.*

Example:

Student.college

Memory Representation



Here:

- *The **static variable college exists only once**.*
- *Both objects refer to the **same variable**.*

4. Final Variables in Java

*In Java, a **final variable** is a variable whose value **cannot be changed once it has been assigned**. It is declared using the keyword **final**. After a value is assigned to a final variable, the variable becomes **constant**, meaning its value remains the same throughout the program execution.*

*Conceptually, final variables are used to represent **fixed values that should not be modified during program execution**. This helps maintain data consistency and prevents accidental changes to important values in a program.*

Final Variables

*When a variable is declared with the final keyword, it can be **assigned a value only once**. After initialization, any attempt to change its value will result in a **compilation error**.*

Example:

```
final int MAX_VALUE = 100;
```

In this example:

- *final indicates that the variable cannot be modified.*
- *MAX_VALUE is the variable name.*
- *100 is the constant value stored in the variable.*

If we try to change the value later:

```
MAX_VALUE = 200;   // Error
```

The compiler will produce an error because a final variable cannot be reassigned.

Purpose of Final Variables

*Final variables are mainly used when a value should remain **constant and unchanged** during the program execution.*

For example:

- *Mathematical constants (PI)*
- *Fixed configuration values*
- *System limits*
- *Identification numbers*

Example:

```
final double PI = 3.14159;
```

Characteristics of Final Variables

Final variables have the following characteristics:

- *Declared using the **final keyword**.*
- *Their value **cannot be modified after initialization**.*
- *They help define **constants in Java programs**.*
- *They improve **program reliability and security**.*

- *They prevent **accidental modification of important values**.*

Example Program

```
public class FinalExample
{
   public static void main(String args[])
   {
      final int speedLimit = 80;

      System.out.println("Speed Limit: " + speedLimit);

      // speedLimit = 100;  // This will cause an error
   }
}
```

Output:

Speed Limit: 80

Here:

- *speedLimit is declared as a final variable.*
- *Its value cannot be changed later in the program.*

2.4 Command Line Arguments in Java

*In Java, **command line arguments** are inputs that are provided to a program **at the time of program execution through the command line**. These arguments allow users to pass values to a program without modifying the program's source code.*

*Command line arguments are received by the **main() method**, which is the entry point of every Java application. The main method contains a parameter that stores the command line arguments in the form of a **string array**.*

public static void main(String[] args)

*Here, the variable args is an array of type **String** that stores all arguments passed during program execution.*

Command Line Arguments

*When a Java program is executed from the command prompt, users can provide values after the program name. These values are called **command line arguments**.*

Example execution:

java Test Hello 25 Java

In this command:

- *Test → program name*
- *Hello, 25, Java → command line arguments*

These values are stored in the args array inside the main() method.

Argument	Array Index	Stored As
Hello	args[0]	First argument
25	args[1]	Second argument
Java	args[2]	Third argument

Thus, command line arguments are stored **sequentially starting from index 0**.

Structure of the Main Method

The main method receives command line arguments using the following syntax:

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

or

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

Explanation:

Keyword	Meaning
public	Accessible from anywhere
static	JVM can call it without creating an object
void	Method does not return any value
main	Entry point of the program
String args[]	Array storing command line arguments

The args parameter can contain **any number of arguments** passed by the user.

Working Mechanism

The working of command line arguments can be understood in the following steps:

1. The program is compiled using the Java compiler.

```
javac Test.java
```

2. The program is executed with arguments.

```
java Test Hello 10 Java
```

3. The Java Virtual Machine (JVM) passes these values to the main() method.
4. The arguments are stored in the **String array args**.
5. The program accesses these values using **array indexing**.

Example Program

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

```
        System.out.println("First Argument: " + args[0]);
        System.out.println("Second Argument: " + args[1]);
        System.out.println("Third Argument: " + args[2]);
    }
}
```

Execution

java CommandLineDemo Apple Banana Mango

Output

First Argument: Apple
Second Argument: Banana
Third Argument: Mango
Here the program prints the arguments passed through the command line.

Characteristics of Command Line Arguments

Command line arguments have several important characteristics:

- *They are passed to the program **during execution time**.*
- *They are stored in the **String array of the main method**.*
- *They allow programs to accept **dynamic input**.*
- *The number of arguments can vary.*
- *They are accessed using **array indexing**.*

Advantages of Command Line Arguments

Command line arguments provide several benefits:

1. *They allow programs to receive **input dynamically at runtime**.*
2. *They eliminate the need to modify the program code for different inputs.*
3. *They make programs more **flexible and reusable**.*
4. *They are useful in **automation, scripting, and batch processing**.*

2.5 Operators in Java

*In Java, **operators** are special symbols used to perform operations on variables and values. They instruct the compiler to perform specific tasks such as **arithmetic calculations, comparisons, logical operations, and value assignments**.*
*Operators work with **operands**, which are the values or variables on which the operation is performed. For example, in the expression:*
int sum = a + b;

- *+ is the **operator***
- *a and b are **operands***

*Operators are essential in programming because they allow programs to **process data, perform calculations, and make decisions during execution**.*

Classification of Operators in Java

Java provides several types of operators that perform different functions. The major categories of operators are:

1. *Arithmetic Operators*
2. *Relational Operators*
3. *Logical Operators*
4. *Assignment Operators*
5. *Increment and Decrement Operators*
6. *Bitwise Operators*
7. *Conditional (Ternary) Operator*
8. *Instanceof Operator*

Each of these operators is explained below.

1. Arithmetic Operators in Java

*In Java, **arithmetic operators** are used to perform **mathematical calculations** on numeric values such as integers and floating-point numbers. These operators are commonly used in programs to carry out operations like **addition, subtraction, multiplication, division, and finding the remainder**.*

*Arithmetic operators work with **operands**, which are the variables or values involved in the calculation. They produce a **numeric result** after performing the operation.*

For example:

```
int sum = a + b;
```

Here:

- *+ is the **arithmetic operator***
- *a and b are **operands***

Types of Arithmetic Operators

Operator	Operation	Description	Example
+	Addition	Adds two values	a + b
-	Subtraction	Subtracts one value from another	a - b
*	Multiplication	Multiplies two values	a * b
/	Division	Divides one value by another	a / b
%	Modulus	Returns the remainder after division	a % b

Example Program

```
public class ArithmeticExample
{
   public static void main(String args[])
   {
      int a = 10;
      int b = 3;

      System.out.println("Addition: " + (a + b));
      System.out.println("Subtraction: " + (a - b));
```

```
        System.out.println("Multiplication: " + (a * b));
        System.out.println("Division: " + (a / b));
        System.out.println("Modulus: " + (a % b));
    }
}
```

Output

Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3
Modulus: 1

2. Relational Operators in Java

*In Java, **relational operators** are used to **compare two values or expressions**. These operators determine the relationship between two operands and return a **Boolean result**, which can be either **true** or **false**.*
*Relational operators are mainly used in **decision-making statements** such as if, if-else, while, and for loops. They help the program decide whether a condition is satisfied or not.*

For example:

```
int a = 10;
int b = 20;

System.out.println(a < b);
```

*In this example, the relational operator < compares a and b. Since 10 is less than 20, the result will be **true**.*

Types of Relational Operators

Operator	Meaning	Description	Example
==	Equal to	Checks if two values are equal	a == b
!=	Not equal to	Checks if two values are not equal	a != b
>	Greater than	Checks if the left value is greater	a > b
<	Less than	Checks if the left value is smaller	a < b
>=	Greater than or equal to	Checks if value is greater or equal	a >= b
<=	Less than or equal to	Checks if value is smaller or equal	a <= b

Example Program

```
public class RelationalExample
{
    public static void main(String args[])
    {
        int a = 10;
        int b = 20;
```

```
        System.out.println(a == b);
        System.out.println(a != b);
        System.out.println(a > b);
        System.out.println(a < b);
        System.out.println(a >= b);
        System.out.println(a <= b);
    }
}
```

Output

```
false
true
false
true
false
true
```

3. Logical Operators in Java

*In Java, **logical operators** are used to **combine or modify logical conditions**. These operators work with **Boolean values** (true or false) and are commonly used in **decision-making statements and loops** such as if, if-else, while, and for.*
*Logical operators allow a program to evaluate **multiple conditions together** and determine whether the overall condition is true or false.*

For example:

```
if(a > 10 && b < 20)
{
   System.out.println("Condition is true");
}
```

*Here, the logical operator && checks whether **both conditions are true**.*

Types of Logical Operators

Operator	***Name***	***Description***	***Example***
&&	Logical AND	Returns true if both conditions are true	(a > b) && (b > c)
`		`	Logical OR
!	Logical NOT	Reverses the logical value	!(a > b)

Truth Table of Logical Operators

Condition A	***Condition B***	***A && B (AND)***	***A || B (OR)***
true	true	true	true
true	false	false	true
false	true	false	true

false	*false*	*false*	*false*

Logical NOT Operator

Condition A	***!A (NOT)***
true	*false*
false	*true*

Example Program

```
public class LogicalExample
{
   public static void main(String args[])
   {
      int a = 10;
      int b = 5;

      System.out.println((a > 5) && (b < 10));
      System.out.println((a > 20) || (b < 10));
      System.out.println(!(a > b));
   }
}
```

Output

```
true
true
false
```

Explanation

- *(a > 5) && (b < 10) → Both conditions are true → result* ***true***
- *(a > 20) || (b < 10) → One condition is true → result* ***true***
- *!(a > b) → Opposite of true → result* ***false***

4. Assignment Operators in Java

In Java, ***assignment operators*** *are used to* ***assign values to variables****. These operators store the value of the right-hand side expression into the variable on the left-hand side.*

The assignment operator is one of the most commonly used operators in Java because variables must be assigned values before they can be used in a program.

For example:

```
int a = 10;
```

Here:

- *= is the* ***assignment operator***
- *10 is the value assigned to variable a*

Types of Assignment Operators

Java provides several assignment operators that combine arithmetic operations with assignment.

Operator	Description	Example	Equivalent Expression
=	Assign value	a = 10	Assigns 10 to a
+=	Add and assign	a += 5	a = a + 5
-=	Subtract and assign	a -= 5	a = a - 5
*=	Multiply and assign	a *= 5	a = a * 5
/=	Divide and assign	a /= 5	a = a / 5
%=	Modulus and assign	a %= 5	a = a % 5

Example Program

```
public class AssignmentExample
{
   public static void main(String args[])
   {
      int a = 10;
      a += 5;
      System.out.println("a += 5 : " + a);
      a -= 3;
      System.out.println("a -= 3 : " + a);
      a *= 2;
      System.out.println("a *= 2 : " + a);
     a /= 4;
      System.out.println("a /= 4 : " + a);
   }
}
```

Output

```
a += 5 : 15
a -= 3 : 12
a *= 2 : 24
a /= 4 : 6
```

5. Increment and Decrement Operators in Java

*In Java, **increment and decrement operators** are special operators used to **increase or decrease the value of a variable by one**. These operators are commonly used in **loops, counters, and iterative processes** where a variable needs to be updated repeatedly.*

*The **increment operator (++)** increases the value of a variable by 1, while the **decrement operator (--)** decreases the value of a variable by 1.*

For example:

```
int a = 5;
a++;
```

*After execution, the value of a becomes **6**.*

Types of Increment and Decrement Operators

Type	*Operator Form*	*Description*	*Example*	*Result*
Pre-Increment	*++a*	*The value of the variable is increased **before it is used** in the expression.*	*int a = 5; int b = ++a;*	*a = 6, b = 6*
Post-Increment	*a++*	*The value of the variable is increased **after it is used** in the expression.*	*int a = 5; int b = a++;*	*a = 6, b = 5*
Pre-Decrement	*--a*	*The value of the variable is decreased **before it is used** in the expression.*	*int a = 5; int b = --a;*	*a = 4, b = 4*
Post-Decrement	*a--*	*The value of the variable is decreased **after it is used** in the expression.*	*int a = 5; int b = a--;*	*a = 4, b = 5*

Example Program

```
public class IncrementExample
{
   public static void main(String args[])
   {
      int a = 5;
      System.out.println("Original value: " + a);
      a++;
      System.out.println("After increment: " + a);
      a--;
      System.out.println("After decrement: " + a);
   }
}
```

Output of the Program

Original value: 5
After increment: 6
After decrement: 5

6. Bitwise Operators in Java

*In Java, **bitwise operators** are used to perform operations on **individual bits of integer data**. These operators work directly on the **binary representation (0s and 1s)** of numbers. Bitwise operators are mainly used in **low-level programming, system programming, and performance optimization**.*
*Since computers internally store numbers in **binary form**, bitwise operators allow programs to manipulate those bits directly.*

*For example, the number **5** in binary is:*

5 = 0101

*And the number **3** in binary is:*

3 = 0011

Bitwise operators perform operations on these binary values.

Types of Bitwise Operators

Operator	***Name***	***Description***	***Example***
&	*Bitwise AND*	*Returns 1 if both bits are 1*	*a & b*
`	*`*	*Bitwise OR*	*Returns 1 if at least one bit is 1*
^	*Bitwise XOR*	*Returns 1 if bits are different*	*a ^ b*
~	*Bitwise NOT*	*Reverses all bits*	*~a*
<<	*Left Shift*	*Shifts bits to the left*	*a << 1*
>>	*Right Shift*	*Shifts bits to the right*	*a >> 1*

Example of Bitwise AND

5 = 0101
3 = 0011

& 0001 = 1

Example of Bitwise OR

5 = 0101
3 = 0011

| 0111 = 7

Example Program

```
public class BitwiseExample
{
   public static void main(String args[])
   {
      int a = 5;
      int b = 3;
      System.out.println("a & b = " + (a & b));
      System.out.println("a | b = " + (a | b));
      System.out.println("a ^ b = " + (a ^ b));
      System.out.println("~a = " + (~a));
   }
}
```

Output

a & b = 1
a | b = 7
a ^ b = 6
~a = -6

7. Conditional (Ternary) Operator in Java

*In Java, the **conditional operator**, also called the **ternary operator**, is used to **evaluate a condition and return one of two values depending on whether the condition is true or false**. It is considered a **short form of the if-else statement** and helps make code shorter and more concise.*
*The ternary operator uses **three operands**, which is why it is called a ternary operator.*

Syntax

condition ? expression1 : expression2;
Explanation:

- ***condition*** *→ the expression that is evaluated*
- ***expression1*** *→ executed if the condition is **true***
- ***expression2*** *→ executed if the condition is **false***

Example

```
public class TernaryExample
{
   public static void main(String args[])
   {
      int marks = 60;

      String result = (marks >= 40) ? "Pass" : "Fail";

      System.out.println(result);
   }
}
```

Pass

Explanation

- *The variable marks is **60**.*
- *The condition (marks >= 40) is **true**.*
- *Therefore, the ternary operator selects **"Pass"** and assigns it to the variable result.*
- *The program prints **Pass**.*

8. instanceof Operator in Java

*In Java, the **instanceof operator** is used to **check whether an object belongs to a particular class or implements a specific interface**. It is mainly used in **object-oriented programming** to verify the type of an object during program execution.*
*The instanceof operator returns a **Boolean value**, which can be either **true** or **false**.*

- *It returns **true** if the object is an instance of the specified class.*

- *It returns **false** if the object does not belong to that class.*

Syntax

object instanceof ClassName

Here:

- ***object** → the reference variable*
- ***ClassName** → the class whose type is being checked*

Example

```
class Student
{
}

public class Test
{
   public static void main(String args[])
   {
      Student s = new Student();

      System.out.println(s instanceof Student);
   }
}
```

Output

true

Explanation

- *s is an object of the **Student** class.*
- *The expression s instanceof Student checks whether s belongs to the **Student** class.*
- *Since it does, the result is **true**.*

Expressions in Java

*In Java, an **expression** is a combination of **variables, constants (literals), operators, and method calls** that are evaluated to produce a **single value**. Expressions are one of the most fundamental parts of programming because they allow a program to **perform calculations, evaluate conditions, and manipulate data**.*

*In simple terms, an expression is **any valid Java statement that results in a value after evaluation**.*

For example:

```
int sum = a + b;
```

Here:

- *a + b is an* ***expression****.*
- *The expression evaluates to a value, which is stored in the variable sum.*

Concept of Expression Evaluation

When the Java compiler executes a program, expressions are evaluated according to specific rules such as:

- ***Operator precedence***
- ***Associativity***
- ***Data type compatibility***

The result of an expression depends on these rules.

Example:

*int result = 10 + 5 * 2;*

Evaluation order:

*10 + (5 * 2)*
10 + 10
= 20

Multiplication is performed first because it has ***higher precedence than addition****.*

Components of an Expression

Components of an Expression

An expression usually consists of the following elements:

- ***Operands*** *→ values or variables used in the expression*
- ***Operators*** *→ symbols that perform operations on operands*
- ***Variables or constants*** *→ data used in calculation*

Example Program

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

```
    {
        int a = 10;
        int b = 5;
        int c = 2;

        int result = a + b * c;

        boolean condition = a > b;

        System.out.println("Result = " + result);
        System.out.println("Condition = " + condition);
    }
}
```

Output

Result = 20

Condition = true

Operator Precedence Table in Java

Precedence Level	***Operators***	***Description***
Highest	*()*	*Parentheses*
	++, --	*Increment / Decrement*
	**, /, %*	*Multiplication, Division, Modulus*
	+, -	*Addition, Subtraction*
	<, >, <=, >=	*Relational Operators*
	==, !=	*Equality Operators*
	&&	*Logical AND*
	`	
Lowest	*=, +=, -=, *=, /=*	*Assignment Operators*

2.7 Type Conversion and Casting in Java

In Java, ***type conversion and casting*** *refer to the process of* ***converting a value from one data type to another data type during program execution****. Since Java is a* ***strongly typed programming language****, every variable must have a specific data type, and the compiler strictly checks type compatibility before performing operations.*

During program execution, situations often arise where values of ***different data types interact in expressions, assignments, or method calls****. In such cases, Java must convert the data from one type to another so that the operation can be performed correctly.*

This process of converting data types is called ***type conversion****.*

Conceptual Understanding of Type Conversion

Every data type in Java has a ***specific size and range of values****.*

Example:

Data Type	***Size***	***Range***
byte	*1 byte*	*-128 to 127*
short	*2 bytes*	*-32,768 to 32,767*
int	*4 bytes*	*-2^{31} to $2^{31}-1$*
long	*8 bytes*	*very large range*

Because of these differences, sometimes Java must ***adjust the type of a value so that it fits into the required data type during operations****.*

Example:

```
int a = 5;
double b = a;
```

Here:

- *a is an integer.*
- *b is a double.*

Java automatically converts 5 into 5.0.

Why Type Conversion is Necessary

Type conversion is required in Java for several reasons:

1. ***Performing operations between different data types***
2. ***Storing values in variables with different types***
3. ***Passing arguments to methods with specific parameter types***

4. ***Preventing type mismatch errors***
5. ***Ensuring accurate mathematical calculations***

Example:

```
int a = 10;
double b = 2.5;

double result = a + b;
```

Evaluation process:

10 → 10.0
10.0 + 2.5 = 12.5

Here Java converts a into a double before performing the operation.

Types of Type Conversion in Java

Java supports two major types of type conversion:

1. ***Implicit Type Conversion (Automatic / Widening)***
2. ***Explicit Type Conversion (Manual / Narrowing)***

---

1. Implicit Type Conversion (Widening Conversion)

*Implicit conversion occurs automatically when **a smaller data type is converted into a larger data type**. This is called widening because the destination type has **greater storage capacity**.*

*This conversion is safe because it **does not cause data loss**.*

Example

```
int number = 25;
double value = number;

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

Output:

25.0

Here:

int → double

The integer value is automatically converted into a floating-point value.

Widening Conversion Hierarchy

Java follows a specific order for automatic type conversion.

byte → short → int → long → float → double

This means:

- *A **byte** value can automatically convert into short, int, long, float, or double.*
- *A **short** can convert into int, long, float, or double.*
- *An **int** can convert into long, float, or double.*

Example:

```
byte a = 10;
int b = a;
double c = b;
```

Here Java performs automatic widening conversions.

---

2. Explicit Type Conversion (Narrowing / Type Casting)

*Explicit type conversion occurs when a **larger data type is converted into a smaller data type**.*

*This is called **narrowing conversion** because the destination type has **less storage capacity**.*

*Because narrowing conversion may cause **loss of information or precision**, the programmer must explicitly specify the conversion using **type casting**.*

Syntax of Type Casting

```
(target_data_type) value
```

Example

```
double number = 9.75;
int result = (int) number;
```

System.out.println(result);

Output:

9

Explanation:

9.75 → 9

The fractional part .75 is lost because integers cannot store decimal values.

Example Program Demonstrating Both Conversions

```
public class ConversionExample
{
   public static void main(String args[])
   {
      int a = 20;
      double b = a;     // widening

      double x = 7.8;
      int y = (int) x;  // narrowing

      System.out.println("Widening Conversion: " + b);
      System.out.println("Narrowing Conversion: " + y);
   }
}
```

Output:

```
Widening Conversion: 20.0
Narrowing Conversion: 7
```

2.8 Control Statements in Java

In Java, ***control statements*** *are used to control the* ***flow of execution of a program****. Normally, a Java program executes statements sequentially from top to bottom. However, in many situations, it becomes necessary to* ***change the normal flow of execution*** *based on conditions or to repeat certain operations multiple times.*

Control statements allow programmers to ***make decisions, repeat tasks, and control the execution order of program instructions****.*

For example, a program may need to:

- *Execute a statement **only if a condition is true***
- ***Repeat a task multiple times***
- ***Skip or stop certain operations***

Control statements help achieve these behaviors.

Types of Control Statements in Java

Control statements in Java are generally classified into three main categories:

1. ***Selection (Decision-Making) Statements***
2. ***Iteration (Looping) Statements***
3. ***Jump (Branching) Statements***

1. Selection (Decision-Making) Statements in Java

*In Java, **selection statements** (also called **decision-making statements**) are used to **control the flow of execution based on conditions**. In a normal program, statements are executed sequentially from top to bottom. However, many real-world problems require a program to **make decisions and choose different actions depending on certain conditions**.*

*Selection statements allow the program to **evaluate a condition and execute specific statements depending on whether the condition is true or false**. These conditions usually produce **Boolean results** (true or false).*

For example, a program may need to decide:

- *Whether a student has **passed or failed***
- *Whether a person is **eligible to vote***
- *Which **menu option** a user selected*

Selection statements make such decisions possible in a program.

Types of Selection Statements in Java

Java provides the following decision-making statements:

1. *if statement*
2. *if–else statement*

3. *else-if ladder*
4. *switch statement*

Each of these statements is explained below.

---

1. The if Statement

*The **if statement** is the simplest decision-making statement in Java. It is used to execute a block of code **only when a specified condition is true**.*

*If the condition evaluates to **true**, the statements inside the if block are executed. If the condition is **false**, the program skips the block and continues with the next statement.*

Syntax

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

Example

```
int age = 20;

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

Explanation

- *The program checks whether age >= 18.*
- *Since the condition is **true**, the message **"Eligible to vote"** is printed.*

2. The if–else Statement

*The **if–else statement** is used when the program needs to choose **between two alternative actions**. One block of code executes if the condition is **true**, and another block executes if the condition is **false**.*

Syntax

```
if(condition)
{
   statements1;
}
```

```
else
{
   statements2;
}
```

Example

```
int number = 10;

if(number % 2 == 0)
{
   System.out.println("Even number");
}
else
{
   System.out.println("Odd number");
}
```

Explanation

- *The program checks whether the number is divisible by 2.*
- *If true → prints* ***Even number***
- *If false → prints* ***Odd number***

3. The else-if Ladder

Sometimes a program needs to choose ***one option from many conditions****. In such cases, the* ***else-if ladder*** *is used.*

In this structure, multiple conditions are checked sequentially. The first condition that evaluates to ***true*** *will execute its block of code, and the remaining conditions are skipped.*

Syntax

```
if(condition1)
{
   statements1;
}
else if(condition2)
{
   statements2;
}
else if(condition3)
{
   statements3;
```

```
}
else
{
   default statements;
}
```

Example

```
int marks = 75;

if(marks >= 90)
{
   System.out.println("Grade A");
}
else if(marks >= 75)
{
   System.out.println("Grade B");
}
else if(marks >= 50)
{
   System.out.println("Grade C");
}
else
{
   System.out.println("Fail");
}
```

Explanation

The program checks conditions one by one:

- *If marks ≥ 90 → Grade A*
- *Else if marks ≥ 75 → Grade B*
- *Else if marks ≥ 50 → Grade C*
- *Otherwise → Fail*

4. The switch Statement

*The **switch statement** is used when a program needs to choose one option from **multiple predefined choices**. It is often used when a variable can have several fixed values.*

*Instead of writing many if-else statements, the switch statement allows a **clean and organized way to handle multiple conditions**.*

Syntax

```
switch(expression)
{
   case value1:
      statements;
      break;

   case value2:
      statements;
      break;

   default:
      statements;
}
```

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");
}
```

Explanation

- *The program checks the value of day.*
- *If the value matches a case, that block is executed.*

2. Iteration (Looping) Statements in Java

*In Java, **iteration statements**, also known as **looping statements**, are used to execute a block of code **repeatedly until a specified condition becomes false**. Loops are very important in*

programming because they allow a program to ***perform repetitive tasks automatically without writing the same code multiple times****.*

In many situations, programs need to repeat certain operations such as ***printing numbers, processing records, calculating sums, or reading data multiple times****. Instead of writing the same instructions repeatedly, iteration statements allow the programmer to define a block of code that will execute multiple times based on a condition.*

For example, if a program needs to print numbers from ***1 to 10****, writing ten separate print statements would be inefficient. Using a loop makes the program* ***shorter, clearer, and easier to maintain****.*

Types of Iteration Statements in Java

Java provides three main looping statements:

1. ***for loop***
2. ***while loop***
3. ***do–while loop***

Each loop controls repetition in a slightly different way.

---

1. The for Loop

The ***for loop*** *is commonly used when the* ***number of iterations is known in advance****. It combines initialization, condition checking, and increment or decrement in a single statement.*

Syntax

```
for(initialization; condition; update)
{
   statements;
}
```

Explanation

- ***Initialization*** *→ initializes the loop variable.*
- ***Condition*** *→ determines whether the loop continues.*
- ***Update*** *→ modifies the loop variable after each iteration.*

Example

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

Output

```
1
2
3
4
5
```

2. The while Loop

*The **while loop** is used when the **number of iterations is not known in advance**. The loop continues executing as long as the specified condition remains **true**.*

*The condition is checked **before executing the loop body**.*

Syntax

```
while(condition)
{
   statements;
}
```

Example

```
int i = 1;

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

Output

```
1
2
3
4
5
```

---

3. The do–while Loop

*The **do–while loop** is similar to the while loop, but the key difference is that the **loop body executes at least once before the condition is checked**.*

*This means the statements inside the loop will execute **even if the condition is false initially**.*

Syntax

```
do
{
   statements;
}
while(condition);
```

Example

```
int i = 1;

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

Output

```
1
2
3
4
5
```

3. Jump (Branching) Statements in Java

*In Java, **jump statements**, also called **branching statements**, are used to **transfer the control of program execution from one part of the program to another**. These statements help alter the normal sequential flow of a program and allow the program to **skip certain statements, exit loops early, or return control from a method**.*

*Normally, Java programs execute statements **sequentially from top to bottom**. However, in many situations a programmer may need to **terminate a loop, skip the current iteration, or exit a method immediately**. Jump statements provide a way to perform these actions.*

*Jump statements are particularly useful when working with **loops and decision-making structures**, where certain conditions may require the program to change its normal flow.*

Types of Jump Statements in Java

Java provides three main jump statements:

1. ***break statement***
2. ***continue statement***
3. ***return statement***

Each of these statements changes the program flow in a different way.

---

1. The break Statement

*The **break statement** is used to **terminate a loop or switch statement immediately**. When the break statement is encountered, the program exits the loop or switch block and continues execution with the next statement after the loop.*

The break statement is commonly used when a specific condition is met and the program no longer needs to continue the loop.

Syntax

```
break;
```

Example

```
for(int i = 1; i <= 5; i++)
{
   if(i == 3)
      break;

   System.out.println(i);
}
```

Output

```
1
2
```

Explanation

- *The loop prints numbers starting from **1**.*
- *When i becomes **3**, the break statement stops the loop.*
- *The program exits the loop immediately.*

---

2. The continue Statement

*The **continue statement** is used to **skip the current iteration of a loop and move directly to the next iteration**. Unlike the break statement, it does not terminate the entire loop; it only skips the current cycle.*

This statement is useful when a certain condition requires skipping a particular iteration without stopping the entire loop.

Syntax

```
continue;
```

Example

```
for(int i = 1; i <= 5; i++)
{
   if(i == 3)
      continue;

   System.out.println(i);
}
```

Output

```
1
2
4
5
```

Explanation

- *When i equals **3**, the continue statement skips that iteration.*
- *The loop continues with the next value of i.*

3. The return Statement

*The **return statement** is used to **terminate the execution of a method and return control to the calling method**. If the method has a return type, the return statement also sends a value back to the caller.*

*The return statement is commonly used in methods when the program needs to **exit the method immediately after producing a result**.*

Syntax

```
return value;
```

or

return;

Example

```
public class Example
{
   static int add(int a, int b)
   {
      return a + b;
   }

   public static void main(String args[])
   {
      int result = add(5, 3);
      System.out.println(result);
   }
}
```

Output

8

Explanation

- *The method add() calculates the sum of two numbers.*
- *The return statement sends the result back to the main() method.*

3. Object-Oriented Programming Basics (Deep Conceptual Explanation)

*In Java, **Object-Oriented Programming (OOP)** is a programming methodology in which a program is organized around **objects and classes rather than individual functions or procedures**. The main goal of object-oriented programming is to represent real-world entities and their interactions in a software system.*

*In traditional programming approaches, programs were designed as a sequence of instructions. However, as software systems became larger and more complex, managing code using only procedures became difficult. Object-oriented programming was introduced to solve this problem by organizing programs into **independent and reusable units called objects**.*

*An **object** in programming represents a real-world entity that contains both **data (attributes)** and **operations (methods)** that can be performed on that data. These objects communicate with each other to perform the tasks required by the program.*

*For example, in a **student management system**, the program may represent students, teachers, and courses as objects. Each object stores its own information and provides functions that operate on that information.*

Basic Idea of Object-Oriented Programming

*The basic idea of object-oriented programming is to model software systems based on **real-world concepts**.*

For example:

Real World Entity	***Attributes (Properties)***	***Behaviors (Functions)***
Student	*name, roll number, marks*	*study(), attendClass()*
Car	*model, color, speed*	*start(), stop(), accelerate()*
Bank Account	*account number, balance*	*deposit(), withdraw()*

*In object-oriented programming, these real-world entities are represented using **classes and objects**.*

Main Components of Object-Oriented Programming

Object-oriented programming is based on several fundamental concepts that help organize and structure programs effectively. These concepts include:

1. *Class*
2. *Object*
3. *Encapsulation*
4. *Inheritance*
5. *Polymorphism*
6. *Abstraction*

*These concepts work together to create **modular, reusable, and maintainable software systems**.*

---

1. Class

*A **class** is a blueprint or template used to create objects. It defines the **properties (variables) and behaviors (methods)** that objects of that class will possess.*

A class does not represent a real object itself. Instead, it describes the structure and behavior that objects created from it will have.

Example:

```
class Student
{
   int rollNo;
   String name;

   void display()
   {
      System.out.println("Student information");
   }
}
```

In this example:

- *Student is a class.*
- *rollNo and name represent the data members.*
- *display() represents the behavior of the class.*

*A class acts as a **blueprint for creating multiple objects**.*

---

2. Object

*An **object** is an instance of a class. It represents a real-world entity and contains actual values for the attributes defined in the class.*

Objects allow the program to access and use the variables and methods defined in the class.

Example:

```
Student s1 = new Student();
```

Here:

- *Student is the class.*
- *s1 is an object.*
- *new is the keyword used to create the object.*

Each object has:

1. ***State** – data stored in variables*
2. ***Behavior** – actions performed by methods*
3. ***Identity** – unique existence in memory*

---

3. Encapsulation

***Encapsulation** refers to the process of **binding data and methods together into a single unit**, which is known as a class. It also involves restricting access to certain components of an object using access modifiers.*

Encapsulation helps protect data from unauthorized access and ensures that it can only be modified through defined methods.

Example concept:

```
Class
 ├── Private Data
 └── Public Methods
```

*Encapsulation improves **data security, modularity, and maintainability**.*

---

4. Inheritance

***Inheritance** is a mechanism through which one class can **inherit the properties and methods of another class**. The class that inherits features is called the **subclass**, while the class being inherited from is called the **superclass**.*

Inheritance allows programmers to reuse existing code and build hierarchical relationships between classes.

Example:

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

class Dog extends Animal
```

```
{
   void bark()
   {
      System.out.println("Dog barks");
   }
}
```

Here the Dog class inherits the eat() method from the Animal class.

*Inheritance promotes **code reuse and logical organization**.*

---

5. Polymorphism

***Polymorphism** means **one interface with multiple forms**. It allows the same method or function to behave differently depending on the context.*

There are two main types of polymorphism:

1. ***Compile-time polymorphism (Method Overloading)***
2. ***Runtime polymorphism (Method Overriding)***

Example:

```
void add(int a, int b)
void add(int a, int b, int c)
```

The method add() performs different operations based on the number of parameters.

*Polymorphism increases **flexibility and extensibility of programs**.*

---

6. Abstraction

***Abstraction** is the process of hiding complex implementation details and showing only the essential features of an object.*

It allows users to interact with objects without needing to understand how the internal processes work.

Example:

```
Car
 ├── start()
 ├── stop()
 └── accelerate()
```

The user can operate the car without knowing the internal working of the engine.

In Java, abstraction is implemented using:

- ***abstract classes***
- ***interfaces***

Advantages of Object-Oriented Programming

Object-oriented programming offers several advantages:

- *Promotes **code reuse through inheritance***
- *Improves **program structure and organization***
- *Simplifies **maintenance and debugging***
- *Supports **modular programming***
- *Represents **real-world systems effectively***
- *Enhances **program scalability***

Class in Java (Very Deep Conceptual Explanation)

*In Java, a **class** is a fundamental concept of object-oriented programming and acts as a **blueprint, template, or prototype used to create objects**. A class defines the **structure and behavior** that objects created from it will possess.*

*A class does not represent a real object directly; instead, it describes **the properties and operations that an object should have**. When a class is defined, it specifies the **data (attributes) and methods (functions)** that represent the characteristics and actions of real-world entities.*

*In object-oriented programming, a class provides a way to **organize data and operations together into a single logical unit**. This combination of data and functions is known as **encapsulation**, which is one of the core principles of OOP.*

Concept of a Class

A class groups together:

- ***Data members (variables)** – store the state or properties of an object*
- ***Member functions (methods)** – define the behavior of an object*

Syntax of a Class

```
class ClassName
{
   // variables
   // methods
}
```

Example Program

```
class Student
{
   int rollNo;
   String name;

   void display()
   {
      System.out.println(rollNo + " " + name);
   }
}

public class Test
{
   public static void main(String args[])
   {
      Student s1 = new Student();

      s1.rollNo = 101;
      s1.name = "Rahul";

      s1.display();
   }
}
```

Output

101 Rahul

Object in Java

*In Java, an **object** is an **instance of a class**. It represents a real-world entity that contains **data (attributes) and behavior (methods)** defined by the class. Objects are the fundamental elements of object-oriented programming because they allow programs to represent and manipulate real-world concepts in software.*

A class defines the structure of an object, while the object represents the ***actual implementation of that structure in memory****. When an object is created, memory is allocated to store the values of its variables and the object can use the methods defined in the class.*
A ***Student object*** *may have:*

- ***State:*** *name, roll number, marks*
- ***Behavior:*** *study(), displayDetails()*
- ***Identity:*** *memory address of the object*

Creation of an Object
Objects in Java are created using the ***new keyword****.*
Syntax
ClassName objectName = new ClassName();
Explanation:

- ***ClassName*** *→ name of the class*
- ***objectName*** *→ reference variable that refers to the object*
- ***new*** *→ keyword used to allocate memory for the object*

Creating Objects from a Class
Once a class is defined, objects can be created using the ***new keyword****.*
Example:
Student s1 = new Student();
Here:

- *Student → class*
- *s1 → object (instance of the class)*

Each object created from the class contains ***its own copy of the variables*** *defined in the class.*
3.3 Constructors in Java
In Java, a ***constructor*** *is a special type of method that is used to* ***initialize objects when they are created****. It is automatically invoked when an object of a class is created. The main purpose of a constructor is to* ***assign initial values to the object's data members and perform any setup required for the object****.*
A constructor has the ***same name as the class*** *and does not have a return type, not even void. When the new keyword is used to create an object, the constructor is called automatically.*
Syntax

```
class ClassName
{
   ClassName()
   {
      // initialization code
   }
}
```

Example

```
class Student
{
   int rollNo;
   String name;

   Student()
   {
      rollNo = 101;
      name = "Ravi";
   }

   void display()
   {
      System.out.println(rollNo + " " + name);
   }
}

public class Test
{
   public static void main(String args[])
   {
      Student s1 = new Student();
      s1.display();
   }
}
```

Output

101 Ravi

Characteristics of Constructors

- *The constructor name must be **the same as the class name**.*
- *It does **not have a return type**.*
- *It is **automatically called when an object is created**.*
- *It is mainly used for **initializing objects**.*

Types of Constructors in Java

*In Java, constructors are special methods used to **initialize objects when they are created**. Depending on how they are defined and used, constructors can be classified into different types. These types allow programmers to initialize objects in different ways and provide flexibility while creating objects.*

The main types of constructors in Java are:

1. ***Default Constructor***
2. ***Parameterized Constructor***
3. ***Copy Constructor (User-defined concept in Java)***

---

1. Default Constructor

*A **default constructor** is a constructor that **does not take any parameters**. If a programmer does not explicitly define any constructor in a class, the Java compiler automatically provides a default constructor.*
*The default constructor initializes the object with **default values**.*

Syntax

```
class ClassName
{
   ClassName()
   {
      // initialization
   }
}
```

Example

```
class Student
{
   int rollNo;
   String name;

   Student()
   {
      rollNo = 100;
      name = "Unknown";
   }

   void display()
   {
      System.out.println(rollNo + " " + name);
   }
}

public class Test
{
   public static void main(String args[])
   {
      Student s1 = new Student();
      s1.display();
   }
}
```

Output

100 Unknown

Characteristics

- *Has **no parameters**.*
- *Automatically called when an object is created.*

- *If no constructor is defined, Java provides one automatically.*

---

2. Parameterized Constructor

*A **parameterized constructor** is a constructor that **accepts arguments (parameters)**. These parameters allow the programmer to initialize objects with **different values at the time of object creation**.*

Parameterized constructors make programs more flexible because different objects can be initialized with different data.

Syntax

```
ClassName(parameters)
{
   // initialization code
}
```

Example

```
class Student
{
   int rollNo;
   String name;

   Student(int r, String n)
   {
      rollNo = r;
      name = n;
   }

   void display()
   {
      System.out.println(rollNo + " " + name);
   }
}

public class Test
{
   public static void main(String args[])
   {
      Student s1 = new Student(101, "Ravi");
      Student s2 = new Student(102, "Anil");

      s1.display();
      s2.display();
   }
}
```

Output

101 Ravi
102 Anil

Characteristics

- *Accepts **arguments during object creation**.*
- *Allows initialization of objects with **different values**.*
- *Improves flexibility of object creation.*

---

3. Copy Constructor

*A **copy constructor** is a constructor used to **create a new object by copying the values of an existing object**. Java does not provide a built-in copy constructor like some other languages, but programmers can define it manually.*

Example

```
class Student
{
   int rollNo;
   String name;

   Student(int r, String n)
   {
      rollNo = r;
      name = n;
   }

   Student(Student s)
   {
      rollNo = s.rollNo;
      name = s.name;
   }

   void display()
   {
      System.out.println(rollNo + " " + name);
   }
}
```

Here the second constructor copies values from another object.

3.4.1 Types of Methods in Java

*In Java, a **method** is a block of code that performs a specific task and is executed when it is called in a program. Methods help organize programs into smaller units and improve **code readability, reusability, and modularity**.*

Based on their characteristics and usage, methods in Java can be classified into several types. These types help programmers define how methods interact with objects and how they handle input and output.

The main types of methods in Java include:

1. ***Instance Methods***
2. ***Static Methods***
3. ***Parameterized Methods***
4. ***Non-Parameterized Methods***
5. ***Methods with Return Values***
6. ***Methods without Return Values***

---

1. Instance Methods

*An **instance method** is a method that belongs to an **object of a class**. These methods can access instance variables directly and are called using an object of the class.*
*Instance methods operate on the **data of specific objects**.*

Example

```
class Student
{
   int rollNo;
   String name;

   void display()
   {
      System.out.println(rollNo + " " + name);
   }
}
```

Method call:

```
Student s1 = new Student();
s1.display();
```

*Here display() is an **instance method** because it is called using the object s1.*

---

2. Static Methods

*A **static method** belongs to the **class rather than an object**. It is declared using the static keyword. Static methods can be called **without creating an object** of the class.*
*Static methods usually perform tasks that are **common to all objects**.*

Example

```
class Test
{
   static void show()
   {
      System.out.println("Static method");
   }
}
```

Method call:

```
Test.show();
```

*Here the method is called using the **class name**.*

3. Parameterized Methods

*A **parameterized method** is a method that accepts **input values called parameters**. These parameters allow methods to perform operations using different data.*

Parameterized methods increase the flexibility and usability of programs.

Example

```
class Calculator
{
   int add(int a, int b)
   {
      return a + b;
   }
}
```

Method call:

```
Calculator c = new Calculator();
System.out.println(c.add(5, 3));
```

Output:

8

---

4. Non-Parameterized Methods

*A **non-parameterized method** does not accept any arguments. It performs operations without requiring external input.*

Example

```
class Message
{
   void display()
   {
      System.out.println("Hello Java");
   }
}
```

Method call:

```
Message m = new Message();
m.display();
```

---

5. Methods with Return Values

*These methods perform a task and **return a value to the calling method**. The returned value can then be used for further calculations or operations.*

Example

```
int square(int n)
{
   return n * n;
}
```

Example call:

System.out.println(square(5));
Output:
25

---

6. Methods without Return Values
*These methods perform an operation but **do not return any value**. Such methods use the keyword void.*
Example

```
void greet()
{
   System.out.println("Welcome to Java");
}
```

3.5 Access Control in Java
*In Java, **access control** refers to the mechanism that determines **how the members of a class such as variables, methods, constructors, and nested classes can be accessed from other parts of a program**. It defines the **visibility and accessibility** of program components.*
*Access control is an important concept in **object-oriented programming** because it helps implement **encapsulation**, which means protecting the internal data of a class and exposing only necessary parts to the outside world.*
*Without access control, every variable and method in a program would be accessible from anywhere, which could lead to **unintended modifications, security issues, and difficult program maintenance**. Access modifiers provide a way to **control how data and methods are used**.*

Concept of Access Control
*In large software systems, different parts of the program must interact with each other in a controlled manner. Some variables and methods should be **accessible only within the class**, while others should be **accessible to subclasses or external classes**.*
Access control allows programmers to specify:

- *Which members can be accessed **within the same class***
- *Which members can be accessed **within the same package***
- *Which members can be accessed **by subclasses***
- *Which members can be accessed **from anywhere***

*These levels of accessibility are controlled using **access modifiers**.*

Access Modifiers in Java
Java provides four types of access modifiers:

1. ***public***
2. ***private***
3. ***protected***
4. ***default (no modifier)***

Each modifier provides a different level of visibility.
3.5.1 Public Access Modifier in Java

*In Java, the **public access modifier** is used to declare class members such as **variables, methods, constructors, and classes** so that they can be **accessed from any other class in any package**. Among all access modifiers, the public modifier provides the **highest level of accessibility**.*
*When a member of a class is declared as public, it becomes **globally accessible within the entire Java program**, regardless of where the accessing class is located. This means that any class, whether it is in the same package or a different package, can use that member without restrictions.*

Concept of Public Access
*In object-oriented programming, some methods or variables need to be **available to other classes so that they can interact with the object**. The public access modifier allows this interaction by making those members visible everywhere.*
*For example, many classes provide **public methods** that allow other classes to use their functionality while keeping internal implementation hidden.*

Syntax

```
public dataType variableName;

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

Example of a public class member:

```
class Student
{
   public int rollNo;

   public void display()
   {
      System.out.println("Student information");
   }
}
```

Here:

- *rollNo is a **public variable***
- *display() is a **public method***

Both can be accessed from other classes.

Example Program

```
class Student
{
   public int rollNo;
```

```
    public void display()
    {
        System.out.println("Roll Number: " + rollNo);
    }
}

public class Test
{
    public static void main(String args[])
    {
        Student s1 = new Student();

        s1.rollNo = 101;
        s1.display();
    }
}
```

Output

Roll Number: 101

Explanation

- *The variable rollNo and method display() are declared as public.*
- *Therefore, they can be accessed directly from another class.*

3.5.2 Private Access Modifier in Java

*In Java, the **private access modifier** is used to restrict the visibility of class members such as **variables, methods, and constructors** so that they can be accessed **only within the same class**. Among all access modifiers, private provides the **most restrictive level of access**.*
*When a variable or method is declared as private, it **cannot be accessed directly from outside the class**, even if the other class is in the same package. This restriction helps protect the internal data of the class and prevents unauthorized modification.*

Concept of Private Access

*In object-oriented programming, classes often contain **internal data that should not be accessed directly by other classes**. If external classes could freely modify these variables, it might lead to incorrect program behavior.*
*The private access modifier ensures that the internal data of a class remains **hidden from other classes**, which is known as **data hiding**.*
*This concept is closely related to **encapsulation**, where the internal state of an object is protected and accessed only through controlled methods.*

Syntax

```
private dataType variableName;

private returnType methodName()
{
```

```
    // method body
}
```

Example:

```
class Student
{
    private int marks;

    private void display()
    {
        System.out.println("Student marks");
    }
}
```

Here:

- *marks is a **private variable***
- *display() is a **private method***

Both can only be accessed inside the Student class.

Example Program

```
class Student
{
    private int marks;

    public void setMarks(int m)
    {
        marks = m;
    }

    public void display()
    {
        System.out.println("Marks: " + marks);
    }
}

public class Test
{
    public static void main(String args[])
    {
        Student s1 = new Student();

        s1.setMarks(85);
        s1.display();
    }
}
```

Output

Marks: 85

Explanation

- *The variable marks is private and cannot be accessed directly.*
- *Instead, public methods setMarks() and display() are used to access it.*

3.5.3 Protected Access Modifier in Java

*In Java, the **protected access modifier** is used to control the accessibility of class members such as **variables, methods, and constructors**. A member declared as protected can be accessed **within the same class, within the same package, and by subclasses (derived classes)** even if those subclasses are located in different packages.*

*The protected modifier provides a **moderate level of accessibility**, which is greater than private but more restricted than public. It is mainly used when designing programs that involve **inheritance**, where a parent class allows its subclasses to access certain members.*

Concept of Protected Access

*In object-oriented programming, classes often form **hierarchical relationships** through inheritance. In such cases, a superclass may contain variables or methods that should be accessible to its subclasses but not necessarily to all other classes.*

*The protected access modifier allows a class to **share its members with its subclasses while still restricting access from unrelated classes**.*

*Thus, protected access helps maintain **controlled sharing of data and methods between related classes**.*

Syntax

```
protected dataType variableName;

protected returnType methodName()
{
   // method body
}
```

Example:

```
class Animal
{
   protected void eat()
   {
      System.out.println("Animal eats food");
   }
}
```

Here the method eat() is declared as protected.

Example Program

```
class Animal
{
   protected void eat()
   {
      System.out.println("Animal eats food");
   }
}

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

   public static void main(String args[])
   {
      Dog d = new Dog();

      d.eat();
      d.bark();
   }
}
```

Output

Animal eats food
Dog barks

Advantages of Protected Access

The protected access modifier provides several benefits:

1. *Allows **controlled access to class members**.*
2. *Supports **inheritance and code reuse**.*
3. *Enables subclasses to use important methods of the superclass.*
4. *Maintains a balance between **security and accessibility**.*
5. *Helps organize programs in hierarchical class structures.*

3.5.4 Default Access Modifier (No Modifier) in Java

*In Java, the **default access modifier** is applied when **no access modifier is specified for a class member**. Because no keyword is used, it is often called **package-private access**.*
*When a variable, method, constructor, or class is declared without any access modifier, it becomes accessible **only within the same package**. Classes that belong to other packages **cannot access these members**.*
*Thus, the default access modifier provides **moderate access control**, allowing access within the same package while restricting access from outside packages.*

Concept of Default Access

*In large Java applications, classes are often organized into **packages**. Sometimes it is necessary for classes within the same package to share data and methods, while preventing classes in other packages from accessing them.*
*The default access modifier allows this type of **package-level access control**. It ensures that members remain accessible only to classes that belong to the same package.*
*Because of this behavior, default access is also known as **package-level visibility**.*

Syntax
Since default access does not use any keyword, the member is declared normally.

```
dataType variableName;

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

Example:

```
class Student
{
   int rollNo;

   void display()
   {
      System.out.println("Student details");
   }
}
```

Here:

- *rollNo is a **default variable***
- *display() is a **default method***

Both are accessible only within the same package.

Example Program

```
class Test
{
   int number = 10;

   void display()
   {
      System.out.println("Number: " + number);
   }
}

public class Demo
{
```

```
    public static void main(String args[])
    {
        Test t = new Test();
        t.display();
    }
}
```

Output

Number: 10

Explanation

- *Both classes belong to the **same package**.*
- *Therefore, the default method display() can be accessed.*

Advantages of Default Access

The default access modifier provides several benefits:

1. *Allows **sharing of data among classes within the same package**.*
2. *Prevents access from classes located in other packages.*
3. *Helps organize programs by controlling package-level visibility.*
4. *Supports modular program design.*

3.6 this Keyword in Java

*In Java, the **this keyword** is a special reference variable that refers to the **current object of the class**. It is automatically created by the Java compiler and is available inside every **instance method and constructor** of a class. The this keyword is used to access the data members and methods of the current object.*

*In object-oriented programming, a class may create multiple objects. Each object has its own copy of instance variables and can call the class methods independently. When a method is invoked by an object, the this keyword inside that method refers to the **object that invoked the method**. Therefore, this represents the **current instance of the class**.*

*The this keyword plays an important role in improving **program clarity, readability, and structure**. It is particularly useful when instance variables and local variables have the same name, or when one constructor needs to call another constructor of the same class.*

Need for this Keyword

*In Java programs, sometimes the **names of instance variables and method parameters are the same**. In such situations, the local variable inside the method hides the instance variable of the class. This creates ambiguity for the compiler.*

*To solve this problem, Java provides the this keyword. It explicitly refers to the **instance variable of the current object**, making the program clear and easy to understand.*

Thus, the this keyword is used to ***avoid ambiguity and access the correct variable or method belonging to the current object****.*

Characteristics of this Keyword

The this keyword has the following important characteristics:

1. *It is a* ***reference variable that refers to the current object****.*
2. *It can be used only inside* ***instance methods and constructors****.*
3. *It cannot be used inside* ***static methods*** *because static methods belong to the class rather than to objects.*
4. *It helps in* ***distinguishing instance variables from local variables****.*
5. *It can be used to* ***call constructors, methods, and pass the current object as an argument****.*

Uses of this Keyword in Java

The this keyword has several important uses in Java programming.

---

1. Referring to the Current Class Object

The simplest use of the this keyword is to refer to the ***current object*** *of a class. When a method is invoked by an object, this inside the method represents that object.*

Example

```
class Demo {
   void show() {
      System.out.println(this);
   }

   public static void main(String args[]) {
      Demo d = new Demo();
      System.out.println(d);
      d.show();
   }
}
```

Explanation

In this program, an object d calls the method show(). Inside the method, this refers to the same object d. Therefore, both statements print the same object reference.

---

2. Accessing Instance Variables

*The this keyword is commonly used to distinguish **instance variables from method parameters** when they have the same name.*

Example

```
class Student {
    int id;
    String name;

    void setData(int id, String name) {
        this.id = id;
        this.name = name;
    }

    void display() {
        System.out.println(id + " " + name);
    }
}
```

Explanation

Here, id and name inside the method are parameters. By writing this.id and this.name, we refer to the instance variables of the class.

---

3. Calling Current Class Methods

The this keyword can be used to call methods of the same class. Although it is optional, it improves the readability of the program.

Example

```
class Test {

    void display() {
        System.out.println("Learning Java");
    }

    void show() {
        this.display();
```

```
    }
}
```

In this example, the method show() calls the method display() using this.display().

---

4. Calling Current Class Constructor

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

*Constructor chaining helps reduce code duplication and improves program organization. One important rule is that this() must always be the **first statement inside the constructor**.*

Example

```
class Student {
    int id;
    String name;

    Student() {
        this(101, "Ravi");
    }

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Here, the default constructor calls the parameterized constructor using this().

---

5. Passing Current Object as an Argument

*The this keyword can be used to pass the **current object as a parameter** to a method.*

Example

```
class Demo {

    void show(Demo obj) {
        System.out.println("Object received");
    }

    void display() {
```

```
        show(this);
    }
}
```

In this example, the current object is passed to the method show() using this.

---

6. Returning the Current Object

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

Example

```
class Test {

    Test show() {
        return this;
    }

    void display() {
        System.out.println("Java Programming");
    }

    public static void main(String args[]) {
        Test t = new Test();
        t.show().display();
    }
}
```

Here, the method show() returns the current object, allowing the next method to be called immediately.

Advantages of Using this Keyword

The this keyword provides several advantages in Java programming:

1. *It helps **remove confusion between instance variables and local variables**.*
2. *It allows **constructors of the same class to call each other**.*
3. *It improves **code readability and clarity**.*
4. *It helps in **passing and returning the current object**.*
5. *It supports **method chaining and better object handling**.*

3.7 Garbage Collection in Java

*Garbage Collection is one of the most important features of Java that provides **automatic memory management**. In Java, when objects are created using the new keyword, memory for those objects is allocated in the **heap memory**. During the execution of a program, many objects are created and used temporarily. After their usage is completed, those objects may no longer be required by the program.*

*If such unused objects remain in memory, they occupy valuable memory space and may eventually lead to **memory wastage or memory overflow**. To avoid this problem, Java provides a mechanism called **Garbage Collection**.*

*Garbage Collection is the process by which the **Java Virtual Machine (JVM) automatically identifies and removes unused objects from heap memory**. The memory that was occupied by these objects is then reclaimed and reused for creating new objects. Because of this automatic process, Java programmers do not need to manually free memory, unlike in languages such as C or C++ where memory management is handled by the programmer.*

*Thus, garbage collection improves **program reliability, memory efficiency, and overall system performance**.*

Concept of Heap Memory and Object Creation

In Java, memory is mainly divided into two important areas:

1. ***Stack Memory***
2. ***Heap Memory***

*The **stack memory** stores local variables and method calls, while the **heap memory** stores objects created during program execution.*

Whenever an object is created using the new keyword, memory is allocated in the heap area. A reference variable stored in the stack points to that object in the heap memory.

For example:

```
class Student {
   int id;
}

public class Test {
   public static void main(String args[]) {
      Student s = new Student();
   }
}
```

*In this example, the object of the Student class is created in **heap memory**, and the reference variable s stored in the stack points to that object.*

*If the reference variable is removed or reassigned, the object becomes **unreachable**, and such an object becomes eligible for garbage collection.*

Working Mechanism of Garbage Collection

*The garbage collector is a component of the **Java Virtual Machine (JVM)** that continuously monitors the objects in heap memory. Its main task is to detect objects that are **no longer referenced by any part of the program**.*

*An object is considered **reachable** if it can still be accessed by a reference variable. If no reference points to that object, it becomes **unreachable**.*

The garbage collector periodically performs the following operations:

1. ***Scanning the heap memory** to identify active objects.*
2. ***Detecting unreachable objects** that are no longer used by the program.*
3. ***Reclaiming the memory occupied by those objects**.*
4. ***Making the reclaimed memory available for future object creation**.*

This process runs automatically in the background without interrupting the program significantly.

Eligibility for Garbage Collection

*An object becomes **eligible for garbage collection** when it is no longer accessible in the program. There are several situations where this can happen.*

1. Nullifying the Reference Variable

If a reference variable is assigned the value null, the object it previously referred to becomes unreachable.

```
class Test {
   public static void main(String args[]) {
      Test t = new Test();
      t = null;
   }
}
```

Here, the object created earlier becomes eligible for garbage collection because the reference variable no longer points to it.

---

2. Reassigning the Reference Variable

If a reference variable is assigned a new object, the old object loses its reference and becomes eligible for garbage collection.

```
class Demo {
   public static void main(String args[]) {
      Demo d1 = new Demo();
      Demo d2 = new Demo();
      d1 = d2;
   }
}
```

In this case, the first object created by d1 becomes unreachable.

---

3. Objects Created Inside Methods

Objects created inside a method become eligible for garbage collection once the method finishes execution.

```
class Example {

   void createObject() {
      Example e = new Example();
   }

}
```

After the method createObject() completes execution, the object e becomes unreachable.

---

4. Island of Isolation

Sometimes objects reference each other but are not reachable from any active part of the program. This situation is called an ***Island of Isolation****.*

```
class Test {
   Test obj;

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

```
        Test t1 = new Test();
        Test t2 = new Test();

        t1.obj = t2;
        t2.obj = t1;

        t1 = null;
        t2 = null;
    }
}
```

Here, both objects refer to each other but are not accessible by any reference variable. Therefore, they become eligible for garbage collection.

Requesting Garbage Collection

*Garbage collection occurs automatically, but programmers can **request** the JVM to perform garbage collection by using certain methods.*

The two commonly used methods are:

- *System.gc()*
- *Runtime.getRuntime().gc()*

Example:

```
class Demo {

    public static void main(String args[]) {

        Demo d1 = new Demo();
        Demo d2 = new Demo();

        d1 = null;
        d2 = null;

        System.gc();
    }
}
```

*However, it is important to note that these methods only **request** garbage collection. The JVM decides when to actually perform the process.*

Role of finalize() Method

Before an object is removed from memory, the JVM may call a method called finalize(). This method allows the object to perform cleanup operations before being destroyed.

Example:

```
class Demo {

   protected void finalize() {
      System.out.println("Object is garbage collected");
   }

   public static void main(String args[]) {

      Demo d = new Demo();
      d = null;

      System.gc();
   }
}
```

When the object becomes eligible for garbage collection, the finalize() method may be executed before the memory is reclaimed.

Advantages of Garbage Collection

Garbage collection provides several advantages in Java programming:

1. *It provides **automatic memory management**.*
2. *It eliminates the need for **manual memory deallocation**.*
3. *It reduces the chances of **memory leaks and memory corruption**.*
4. *It improves **program reliability and security**.*
5. *It allows programmers to focus more on **program logic rather than memory management**.*

Limitations of Garbage Collection

Despite its advantages, garbage collection has some limitations:

1. *Programmers **cannot control the exact time** when garbage collection occurs.*
2. *Calling System.gc() does not guarantee immediate garbage collection.*
3. *Garbage collection may sometimes cause **temporary performance overhead**.*

3.8 Nested Classes in Java

*In Java, a **nested class** is a class that is declared **inside another class**. The class that contains another class is called the **outer class**, and the class defined within it is called the **nested class**.*

*Nested classes are used to logically group classes that are **closely related to each other**. By placing one class inside another class, the programmer can improve the **organization, readability, and maintainability of the program**. Nested classes also help in **encapsulation** because they can access the members of the outer class, including private members.*

*In Java, nested classes allow the program structure to be more **modular and secure**, especially when a class is useful only within another class and does not need to be accessed from outside.*

Need for Nested Classes

In large programs, sometimes a class is useful only for supporting another class. If such helper classes are defined separately, they may unnecessarily increase the complexity of the program.

To avoid this problem, Java allows classes to be defined inside other classes. This helps in:

- *Grouping logically related classes together*
- *Improving code readability and structure*
- *Increasing encapsulation and data hiding*
- *Reducing namespace conflicts*

Thus, nested classes help organize large programs more efficiently.

Types of Nested Classes in Java

Java supports two main types of nested classes:

1. ***Static Nested Class***
2. ***Inner Class (Non-static Nested Class)***

The inner class is further divided into several types.

Nested Classes

```
|
├── Static Nested Class
|
└── Inner Classes
    |
    ├── Member Inner Class
    ├── Local Inner Class
    └── Anonymous Inner Class
```

1. Static Nested Class

*A **static nested class** is a class that is declared inside another class using the **static keyword**. Since it is static, it behaves similarly to a **static member of the outer class**.*

*A static nested class is associated with the **outer class itself rather than with an object of the outer class**. Therefore, an object of the outer class is **not required** to create an instance of a static nested class.*

Concept

*The static nested class can access **only the static members of the outer class directly**. It cannot directly access the non-static members of the outer class unless an object of the outer class is created.*

This type of class is mainly used when the nested class is related to the outer class but does not require access to the instance variables of the outer class.

Example

```
class Outer {

    static int x = 10;

    static class Nested {

        void display() {
            System.out.println("Value of x: " + x);
        }
    }

    public static void main(String args[]) {
        Outer.Nested obj = new Outer.Nested();
        obj.display();
```

```
    }
}
```

Explanation

*In this program, the class Nested is declared as a **static class inside the class Outer**. Since it is static, it can be accessed using the syntax Outer.Nested. The nested class directly accesses the static variable x of the outer class.*

Characteristics of Static Nested Class

- *It is declared using the **static keyword**.*
- *It can access **only static members** of the outer class directly.*
- *It does **not require an object of the outer class**.*
- *It is accessed using the syntax **OuterClass.NestedClass**.*
- *It behaves similarly to a **static member of a class**.*

---

2. Inner Class (Non-static Nested Class)

*An **inner class** is a nested class that is declared **without using the static keyword**. Unlike static nested classes, inner classes are associated with the **instance (object) of the outer class**.*

*This means that an inner class **cannot exist independently** without an object of the outer class. It has access to **all members of the outer class**, including private members.*

Concept

*The inner class forms a strong relationship with the outer class. Because of this relationship, it can directly access the **instance variables, methods, and private members** of the outer class.*

*Inner classes are useful when a class is required only for **supporting the functionality of the outer class**.*

Example

```
class Outer {

    int num = 20;

    class Inner {

        void display() {
```

```
            System.out.println("Value of num: " + num);
        }
    }

    public static void main(String args[]) {
        Outer obj = new Outer();
        Outer.Inner in = obj.new Inner();
        in.display();
    }
}
```

Explanation

In this example, the class Inner is defined inside the class Outer. To create an object of the inner class, we must first create an object of the outer class. The inner class can directly access the instance variable num of the outer class.

Characteristics of Inner Class

- *It is declared **without the static keyword**.*
- *It requires an **object of the outer class**.*
- *It can access **all members of the outer class**, including private members.*
- *It is tightly associated with the **instance of the outer class**.*

Further Types of Inner Classes in Java

*An **inner class** is a non-static nested class that is defined inside another class. Inner classes have a strong relationship with the outer class and can directly access its **instance variables, methods, and even private members**.*

*To provide more flexibility in program design, Java divides inner classes into **three different types based on their place of declaration and scope**. These types help programmers use classes only where they are required, which improves **program structure, security, and encapsulation**.*

The three types of inner classes are:

1. ***Member Inner Class***
2. ***Local Inner Class***
3. ***Anonymous Inner Class***

---

1. Member Inner Class

*A **member inner class** is a class that is declared **inside the outer class but outside any method, constructor, or block**. It behaves like a normal member of the outer class, similar to instance variables and methods.*

*Because it is associated with an object of the outer class, a member inner class **cannot exist independently**. To create an object of the member inner class, an object of the outer class must first be created.*

Concept

*The member inner class can directly access **all members of the outer class**, including private variables and methods. This is possible because the inner class is considered part of the outer class.*

*This type of class is useful when a class is **closely related to the outer class and is used repeatedly inside that class**.*

Example

```
class Outer {

    int x = 50;

    class Inner {

        void display() {
            System.out.println("Value of x: " + x);
        }
    }

    public static void main(String args[]) {

        Outer obj = new Outer();
        Outer.Inner in = obj.new Inner();

        in.display();
    }
}
```

Explanation

In this example, the class Inner is declared inside the class Outer. The inner class can directly access the variable x of the outer class. To create the object of the inner class, the object of the outer class must be created first.

2. Local Inner Class

*A **local inner class** is a class that is declared **inside a method of the outer class**. Its scope is limited only to that particular method in which it is defined.*

*This type of class is useful when a class is required only for a **specific task inside a method** and is not needed elsewhere in the program.*

Concept

The local inner class can access the members of the outer class as well as the local variables of the method (if they are effectively final). However, it cannot be accessed outside the method in which it is declared.

Thus, the lifetime and visibility of the local inner class are limited to that method.

Example

```
class Outer {

    void display() {

        class Inner {

            void show() {
                System.out.println("This is a Local Inner Class");
            }
        }

        Inner obj = new Inner();
        obj.show();
    }

    public static void main(String args[]) {

        Outer o = new Outer();
        o.display();
    }
}
```

Explanation

In this example, the class Inner is declared inside the method display(). Therefore, it can be used only inside that method and cannot be accessed outside it.

3. Anonymous Inner Class

*An **anonymous inner class** is a class that **does not have a name**. It is declared and instantiated **at the same time in a single statement**.*

*Anonymous inner classes are generally used when a class is required **only once** and there is no need to reuse it elsewhere in the program.*

Concept

*Instead of defining a separate class, Java allows programmers to create a class without a name and directly provide the implementation of methods. Anonymous classes are commonly used with **interfaces or abstract classes**.*

This reduces the need for writing separate class files and makes the code shorter and more efficient.

Example

```
abstract class Animal {

    abstract void sound();
}

class Test {

    public static void main(String args[]) {

        Animal a = new Animal() {

            void sound() {
                System.out.println("Dog Barks");
            }
        };

        a.sound();
    }
}
```

Explanation

In this example, an anonymous class is created that extends the abstract class Animal. The implementation of the sound() method is provided directly at the time of object creation.

3.10 String in Java

*A **String** in Java is a sequence of characters used to represent textual information such as names, messages, addresses, and other forms of text data. Strings are one of the most commonly used data types in programming because most applications involve handling text.*

*In Java, a String is **not a primitive data type**. Instead, it is an **object of the String class**, which is defined in the **java.lang package**. Since the java.lang package is automatically imported in every Java program, the String class can be used directly without writing an import statement.*

*A string consists of a series of characters enclosed in **double quotation marks (" ")**. When a string is created in Java, an object of the String class is created to store the sequence of characters.*

Example:

String message = "Welcome to Java";

In this example, "Welcome to Java" is a string literal and message is a reference variable that refers to the string object.

*Strings play a very important role in Java programming because they are used in **input and output operations, file handling, web applications, and user interface development**.*

Concept of Strings in Java

*In Java, strings are treated as **objects rather than simple variables**. When a string is created, the Java Virtual Machine (JVM) allocates memory for that string object in the **heap memory**.*

*One of the most important characteristics of strings in Java is that **strings are immutable**. Immutability means that once a string object is created, **its value cannot be changed**. If any modification is performed on a string, Java does not change the original string; instead, it creates a **new string object with the modified value**.*

Example:

```
String str = "Hello";
str = str + " Java";
```

In this case, the original string "Hello" is not modified. Instead, a new string "Hello Java" is created and the reference variable str now refers to the new object.

*This property of immutability provides **security, reliability, and efficient memory usage**.*

Ways of Creating Strings in Java

*In Java, strings can be created in **two main ways**.*

1. Creating String Using String Literal

A string literal is created by directly assigning a sequence of characters inside double quotation marks.

Example:

String s1 = "Java Programming";

When a string literal is created, Java stores it in a special memory area called the ***String Constant Pool (SCP)****. If another string with the same value already exists in the pool, Java reuses the existing object instead of creating a new one. This mechanism helps in* ***saving memory and improving performance****.*

---

2. Creating String Using new Keyword

A string can also be created by explicitly creating an object of the String class using the new keyword.

Example:

String s2 = new String("Java Programming");

In this method, Java always creates a ***new string object in heap memory****, even if another object with the same value already exists. Therefore, this method consumes* ***more memory compared to string literals****.*

Important Characteristics of Strings

Strings in Java have several important characteristics:

1. *Strings are* ***objects of the String class****.*
2. *They are stored in* ***heap memory****.*
3. *Strings are* ***immutable****, meaning their values cannot be modified after creation.*
4. *Java uses a* ***String Constant Pool*** *to store string literals efficiently.*
5. *The String class provides many* ***built-in methods*** *for manipulating text.*

Method	***Description***	***Example***	***Output***
length()	*Returns the number of characters in the string.*	*"Java".length()*	*4*

Method	Description	Example	Output
charAt(int index)	Returns the character at the specified index.	"Java".charAt(2)	v
concat(String s)	Joins two strings together.	"Java".concat(" Programming")	Java Programming
equals(Object obj)	Compares two strings and returns true if they are equal.	"Java".equals("Java")	true
equalsIgnoreCase(String s)	Compares two strings ignoring case differences.	"java".equalsIgnoreCase("JAVA")	true
toUpperCase()	Converts all characters of the string to uppercase.	"java".toUpperCase()	JAVA
toLowerCase()	Converts all characters of the string to lowercase.	"JAVA".toLowerCase()	java
trim()	Removes leading and trailing spaces from a string.	" Java ".trim()	Java
substring(int beginIndex)	Returns part of the string starting from the given index.	"Java".substring(1)	ava
substring(int begin, int end)	Returns characters between the specified indexes.	"Java".substring(1,3)	av

Method	*Description*	*Example*	*Output*
replace(char old, char new)	*Replaces characters in the string.*	*"Java".replace('a','o')*	*Jovo*
contains(CharSequence s)	*Checks whether the string contains a specified sequence.*	*"Java".contains("av")*	*true*
indexOf(String s)	*Returns the index of the first occurrence of a character or substring.*	*"Java".indexOf("a")*	*1*
lastIndexOf(String s)	*Returns the index of the last occurrence of a character.*	*"Java".lastIndexOf("a")*	*3*
isEmpty()	*Checks whether the string is empty.*	*"".isEmpty()*	*true*

3.11 StringBuffer

*StringBuffer is a class in Java used to create **mutable strings**. Unlike the String class, the contents of a StringBuffer object **can be modified after it is created** without creating a new object.*

*The StringBuffer class is defined in the **java.lang package** and is mainly used when a string needs to be **modified frequently**, such as adding, deleting, or changing characters.*

Concept of Mutable Strings

*A mutable object is an object whose value can be changed after creation. StringBuffer allows modification of the same object, which makes it more **memory efficient** when performing many string operations.*

Example:

```
StringBuffer sb = new StringBuffer("Java");
sb.append(" Programming");
System.out.println(sb);
```

Output:

Java Programming

Here, the original object is modified instead of creating a new one.

Method	*Description*	*Example*	*Output*
append()	*Adds text or characters at the end of the string.*	*StringBuffer sb = new StringBuffer("Hello"); sb.append(" Java");*	*Hello Java*
insert(int offset, String str)	*Inserts a string at a specific position in the string.*	*StringBuffer sb = new StringBuffer("Helo"); sb.insert(3,"l");*	*Hello*
delete(int start, int end)	*Removes characters from the string between the specified indexes.*	*StringBuffer sb = new StringBuffer("Hello"); sb.delete(1,4);*	*Ho*
reverse()	*Reverses the sequence of characters in the string.*	*StringBuffer sb = new StringBuffer("Hello"); sb.reverse();*	*olleH*
replace(int start, int end, String str)	*Replaces characters between specified indexes with another string.*	*StringBuffer sb = new StringBuffer("Hillo"); sb.replace(1,2,"e");*	*Hello*

Example Program

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

      StringBuffer sb = new StringBuffer("Hello");

      sb.append(" Java");
      System.out.println(sb);

      sb.reverse();
      System.out.println(sb);
   }
}
```

Characteristics of StringBuffer

- *It is **mutable**.*

- *It allows modification of the same object.*
- *It is **thread-safe**, meaning multiple threads can use it safely.*
- *It is useful when strings are modified frequently.*

3.12 StringBuilder

*StringBuilder is another class used for creating **mutable strings** in Java. It was introduced in **Java 1.5** to improve performance when performing many string operations.*

*Like StringBuffer, the StringBuilder class allows modification of strings without creating new objects. However, it is **not synchronized**, which means it is **not thread-safe**.*

*Because of this, StringBuilder is generally **faster than StringBuffer** and is used when thread safety is not required.*

Example

```
StringBuilder sb = new StringBuilder("Java");
sb.append(" Language");
System.out.println(sb);
```

Output:

Java Language

Important Methods of StringBuilder

Method	***Description***	***Example***	***Output***
append()	*Adds characters or a string to the end of the existing string.*	*StringBuilder sb = new StringBuilder("Java"); sb.append(" Programming");*	*Java Programming*
insert(int offset, String str)	*Inserts a string at a specified position in the string.*	*StringBuilder sb = new StringBuilder("Jav"); sb.insert(3,"a");*	*Java*
delete(int start, int end)	*Removes characters between the specified start and end indexes.*	*StringBuilder sb = new StringBuilder("Java"); sb.delete(1,3);*	*Ja*
reverse()	*Reverses the characters in the string.*	*StringBuilder sb = new StringBuilder("Java"); sb.reverse();*	*avaJ*
replace(int start, int end, String str)	*Replaces characters between given indexes with another string.*	*StringBuilder sb = new StringBuilder("Jaba"); sb.replace(2,4,"va");*	*Java*

Example Program

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

      StringBuilder sb = new StringBuilder("Java");

      sb.append(" Programming");
      System.out.println(sb);

      sb.insert(4, " Language");
      System.out.println(sb);

      sb.delete(4,13);
      System.out.println(sb);

      sb.reverse();
      System.out.println(sb);
   }
}
```

Output

Java Programming
Java Language Programming
Java Programming
gnimmargorP avaJ

UNIT – II

1.Inheritance

1.1 Inheritance Concept in Java

*Inheritance is one of the most important concepts in **Object-Oriented Programming (OOP)**. In Java, inheritance allows one class to acquire the properties and behaviors of another class. The class that provides the properties and methods is called the **parent class** (or **superclass**), and the class that inherits those features is called the **child class** (or **subclass**).*

*The main idea behind inheritance is **code reusability**. Instead of writing the same code again in multiple classes, a new class can reuse the existing code from another class. This reduces duplication of code and makes programs easier to maintain and understand.*

*In Java, inheritance is implemented using the **extends** keyword. When a subclass extends a superclass, it automatically gets access to the variables and methods of the parent class. The subclass can also add its own new properties and methods, or it can modify the behavior of the inherited methods.*

*Inheritance also helps in creating a **hierarchical relationship between classes**. It represents an **"is-a" relationship**. For example, if we have a class called Animal, a class called Dog can inherit from it because a dog **is an animal**. In this case, the Dog class can use the common features of the Animal class such as eating, sleeping, and moving.*

*Another advantage of inheritance is that it supports **method overriding**, where a subclass can provide its own implementation of a method that is already defined in the parent class. This allows programmers to create more specific behavior for different objects while still maintaining the common structure.*

*Thus, inheritance plays a very important role in Java programming. It helps in building programs in a **structured, reusable, and organized way**, which is one of the main goals of object-oriented programming.*

1.2 Types of Inheritance in Java

Java supports the following types of inheritance:

1. ***Single Inheritance***
   *In this type, **one child class inherits from one parent class**.*
2. ***Multilevel Inheritance***
   *In this type, **a class inherits from another class, and that class is inherited by another class**, forming a chain.*
3. ***Hierarchical Inheritance***
   *In this type, **multiple child classes inherit from a single parent class**.*
4. ***Multiple Inheritance (through Interfaces)***
   *In this type, **one class can inherit features from more than one interface**.*
5. ***Hybrid Inheritance (using Interfaces)***
   *This type is a **combination of two or more types of inheritance**, usually implemented using interfaces in Java.*

1. Single Inheritance in Java

*Single inheritance is a type of inheritance where **one child class inherits the properties and behaviors of only one parent class**. It is the simplest form of inheritance used in Java. In this concept, a new class is created based on an existing class, so that the new class can use the features that are already defined in the parent class.*

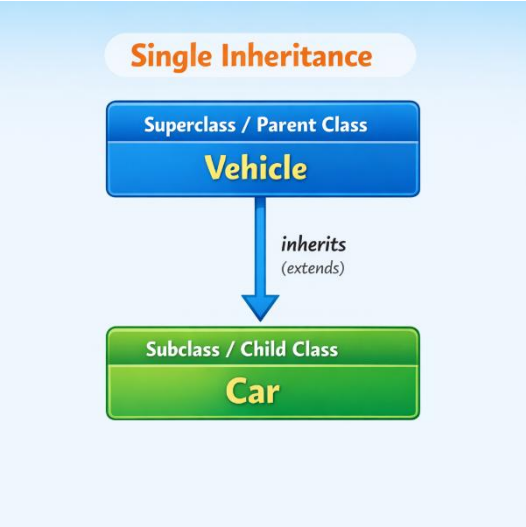
*The parent class is also called the **superclass**, and the child class is called the **subclass**. The subclass automatically gains access to the methods and variables of the superclass. This allows the subclass to use the existing functionality of the parent class and also add new features of its own.*

*The main idea behind single inheritance is **reusability of code**. Instead of writing the same code again in different classes, a programmer can create a common class with shared*

features and allow other classes to inherit those features. This makes the program shorter, easier to manage, and more efficient.

*In Java, single inheritance is implemented using the **extends keyword**. When one class extends another class, it forms a **parent–child relationship** between them. This relationship is often described as an **"is-a" relationship**. For example, if there is a class called **Animal**, another class called **Dog** can inherit from it because a dog **is an animal**. The Dog class can use the common behaviors of the Animal class such as eating or sleeping, and it can also define its own behavior like barking.*

*Single inheritance also helps in building programs in a **structured and logical way**. It makes the relationship between classes clear and improves the readability of the program. Because of its simplicity, single inheritance is widely used in many Java applications.*



Advantages of Single Inheritance

1. ***Code Reusability***
   The child class can reuse the methods and variables of the parent class, so the same code does not need to be written again.
2. ***Simple and Easy to Understand***
   Since only one parent class is involved, the structure of the program is clear and easy to follow.
3. ***Better Code Organization***
   It helps in organizing related classes in a logical parent–child relationship.
4. ***Easy Program Maintenance***
   If changes are made in the parent class, they can automatically apply to the child class, making the program easier to update.

Disadvantages of Single Inheritance

1. ***Limited Inheritance***
   A class can inherit from only one parent class, so it cannot directly use features from multiple classes.

2. ***Less Flexibility***
   If additional features are needed from other classes, they cannot be inherited directly in single inheritance.
3. ***Dependency on Parent Class***
   The child class depends on the parent class, so changes in the parent class may affect the child class.

Java Program Example

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

class Car extends Vehicle {
    void drive() {
        System.out.println("Car is driving");
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.start();   // inherited from Vehicle
        c.drive();   // method of Car class
    }
}
```

Explanation

*In this program, the **Car class extends the Vehicle class**. This means the Car class inherits the start() method from the Vehicle class. The Car class also has its own method called drive(). When an object of the Car class is created, it can access both the inherited method and its own method.*

*This is an example of **single inheritance**, because **one child class (Car) inherits the properties and behavior of one parent class (Vehicle)**.*

---

2. Multilevel Inheritance in Java

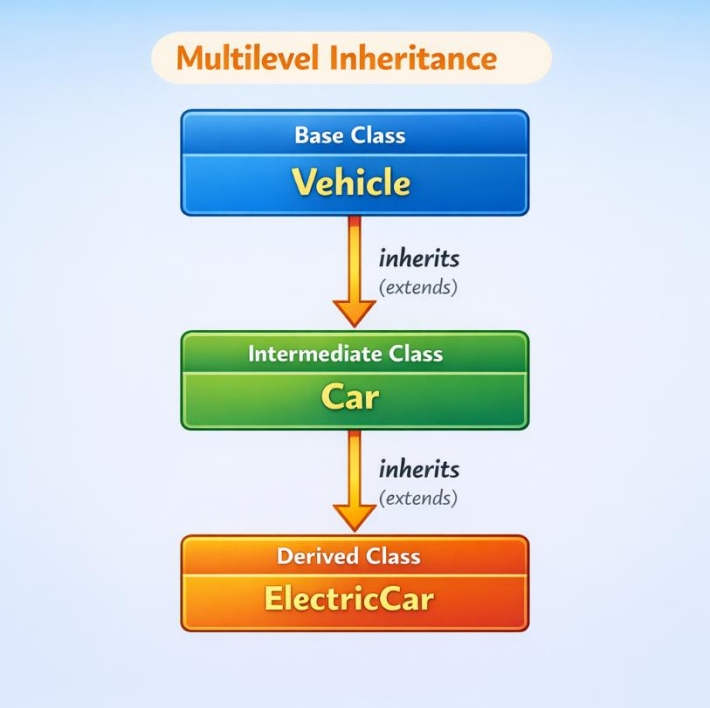
*Multilevel inheritance is a type of inheritance in which **a class is derived from another class, and then another class is derived from that derived class**. In simple terms, inheritance takes place in **multiple levels**, forming a chain of classes. In this structure, a child class becomes the parent class for another class.*

*In multilevel inheritance, the first class is called the **base class or superclass**, the second class acts as an **intermediate class**, and the last class is known as the **derived class or subclass**. The final class in the hierarchy can access the properties and methods of all the classes that come before it. This means that the features of the base class are passed through several levels of inheritance.*

*The main goal of multilevel inheritance is to **reuse existing code and build programs step by step**. Each class can add new functionality while still using the features of the previous classes. This helps programmers develop complex programs in a more organized and systematic way.*

*In Java, multilevel inheritance is implemented using the **extends keyword**. Each class extends the class above it, creating a chain-like relationship. This type of inheritance also shows a **"is-a" relationship across multiple levels**. For example, if there is a class called **Vehicle**, a class called **Car** can inherit from Vehicle, and another class called **ElectricCar** can inherit from Car. In this case, an ElectricCar is a Car, and a Car is a Vehicle.*

*Therefore, multilevel inheritance helps in creating programs that are **well structured, reusable, and easier to expand** as the program grows.*



- ***Vehicle** is the **Base Class (Superclass)**.*
- ***Car** is the **Intermediate Class** because it inherits from Vehicle and also acts as a parent for the next class.*
- ***ElectricCar** is the **Derived Class (Subclass)**.*

*The diagram shows that **ElectricCar inherits features from Car, and Car inherits features from Vehicle**.*

*Therefore, the **ElectricCar class can access the properties and methods of both Vehicle and Car**.*

*This structure represents **Multilevel Inheritance**, where inheritance occurs through **multiple levels forming a chain of classes**.*

Advantages of Multilevel Inheritance

1. ***Code Reusability***
   The derived classes can reuse the methods and properties of the parent classes, which reduces code duplication.
2. ***Better Program Organization***
   It helps in organizing classes in a logical order or hierarchy.
3. ***Easy Extension of Programs***
   New classes can be added easily without modifying the existing classes.
4. ***Improved Maintainability***
   Changes made in the base class can automatically be reflected in the derived classes.

Disadvantages of Multilevel Inheritance

1. ***Increased Complexity***
   As the number of inheritance levels increases, the program structure may become difficult to understand.
2. ***Difficult Debugging***
   Errors may be harder to trace because methods and properties are inherited through multiple levels.
3. ***Tight Dependency***
   The lower-level classes depend on the higher-level classes, so changes in parent classes may affect many child classes.

Java Program for Multilevel Inheritance

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

class Car extends Vehicle {
    void drive() {
        System.out.println("Car is driving");
    }
}

class ElectricCar extends Car {
    void charge() {
        System.out.println("Electric car is charging");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        ElectricCar e = new ElectricCar();

        e.start();   // inherited from Vehicle
        e.drive();   // inherited from Car
        e.charge();  // method of ElectricCar
    }
}
```

Explanation

*In this program, **Vehicle** is the base class, **Car** is the intermediate class that inherits from Vehicle, and **ElectricCar** is the derived class that inherits from Car.*

*The object of the **ElectricCar** class can access methods from **all three classes**. It can use start() from Vehicle, drive() from Car, and charge() from ElectricCar.*

*This shows **Multilevel Inheritance**, where inheritance occurs in **multiple levels forming a chain of classes**.*

3. Hierarchical Inheritance in Java

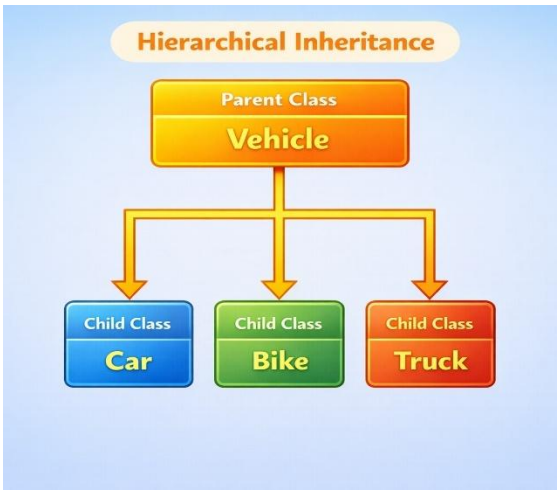
*Hierarchical inheritance is a type of inheritance in which **multiple child classes inherit the properties and behaviors from a single parent class**. In this structure, one superclass acts as a common base, and several subclasses are derived from it. Each child class can use the features of the parent class while also adding its own specific properties and methods.*

*The main purpose of hierarchical inheritance is to **reuse common code in multiple classes**. Instead of writing the same properties and methods separately in many classes, they can be defined once in a parent class. Then several child classes can inherit those features and extend them according to their own needs.*

*In Java, hierarchical inheritance is implemented using the **extends keyword**. Different subclasses extend the same parent class. As a result, all child classes can access the common features defined in the parent class, but each subclass can also include its own unique functionality.*

*Hierarchical inheritance also represents an **"is-a" relationship**. For example, if there is a class called **Vehicle**, classes such as **Car**, **Bike**, and **Truck** can inherit from it. In this case, a Car is a Vehicle, a Bike is a Vehicle, and a Truck is also a Vehicle. All these classes share the common features of the Vehicle class, but they can also have their own specific behaviors.*

*Therefore, hierarchical inheritance helps in creating programs that are **well organized, reusable, and easy to manage**, especially when several classes share similar characteristics.*



- ***Vehicle*** *is the* ***Parent Class (Superclass)****.*
- ***Car****,* ***Bike****, and* ***Truck*** *are* ***Child Classes (Subclasses)****.*
- *All child classes* ***inherit the properties and methods of the Vehicle class****.*
- *Each child class can also define its own additional features.*

This diagram shows ***Hierarchical Inheritance****, where* ***multiple child classes inherit from a single parent class****.*

Advantages of Hierarchical Inheritance

1. ***Code Reusability***
   Common features are written once in the parent class and reused by multiple child classes.
2. ***Better Organization***
   Classes can be arranged in a clear and logical structure.
3. ***Reduced Code Duplication***
   Repeated code is minimized because shared functionality is defined in one place.
4. ***Easy Maintenance***
   If changes are made in the parent class, they can automatically affect all child classes.

Disadvantages of Hierarchical Inheritance

1. ***Dependency on Parent Class***
   All child classes depend on the parent class, so changes in the parent class may affect them.
2. ***Complexity with Many Subclasses***
   If there are many subclasses, the program structure may become difficult to manage.
3. ***Limited Flexibility***
   A subclass inherits only from one parent class, so it cannot directly inherit features from other classes.

Java Program for Hierarchical Inheritance

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

class Car extends Vehicle {
    void drive() {
        System.out.println("Car is driving");
    }
}

class Bike extends Vehicle {
    void ride() {
        System.out.println("Bike is riding");
    }
}

public class Main {
    public static void main(String[] args) {

        Car c = new Car();
        c.start();   // inherited from Vehicle
        c.drive();   // method of Car

        Bike b = new Bike();
        b.start();   // inherited from Vehicle
        b.ride();    // method of Bike
    }
}
```

Explanation

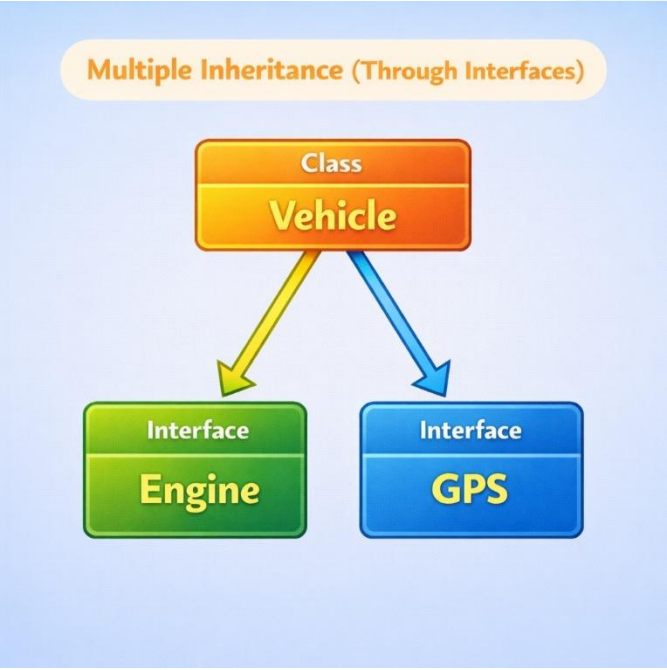
*In this program, **Vehicle** is the **parent class**. Two classes, **Car** and **Bike**, inherit from the Vehicle class. Both child classes can use the start() method defined in the Vehicle class. At the same time, each class has its own specific method. The **Car** class has the drive() method, and the **Bike** class has the ride() method.*

*This is called **Hierarchical Inheritance** because **multiple child classes (Car and Bike) inherit from a single parent class (Vehicle)**.*

4. Multiple Inheritance in Java (Through Interfaces)

*Multiple inheritance is a type of inheritance where **a class inherits the properties and behaviors from more than one parent source**. In many programming languages, a class can directly inherit from multiple classes. However, **Java does not support multiple inheritance***

using classes *because it may create ambiguity and confusion when two parent classes contain methods with the same name.*
To solve this problem, Java provides ***multiple inheritance through interfaces****. An interface is a special type of structure in Java that contains method declarations without implementation. A class can* ***implement more than one interface****, which allows it to inherit the behaviors of multiple interfaces at the same time.*
When a class implements multiple interfaces, it must provide the ***implementation for all the methods declared in those interfaces****. This helps in combining different functionalities into a single class while maintaining clarity and avoiding conflicts.*
Multiple inheritance through interfaces improves ***code reusability and flexibility*** *in program design. It allows developers to build programs in a modular way by separating different functionalities into different interfaces and then combining them in a class.*



For example, a ***Vehicle*** *can have different capabilities such as* ***Engine operations*** *and* ***Navigation features****. These features can be defined in two interfaces called* ***Engine*** *and* ***GPS****. A class called* ***Car*** *can implement both interfaces. In this case, the Car class inherits the features of both Engine and GPS interfaces.*
Thus, multiple inheritance through interfaces helps in ***combining multiple functionalities in a single class while avoiding the problems associated with multiple class inheritance****.*

Advantages of Multiple Inheritance (Using Interfaces)

1. ***Code Reusability*** *– A class can use features from multiple interfaces.*
2. ***Flexibility*** *– It allows combining different functionalities in one class.*
3. ***Avoids Ambiguity*** *– Interfaces prevent conflicts that occur in multiple class inheritance.*
4. ***Better Program Design*** *– Programs become modular and well structured.*

Disadvantages of Multiple Inheritance (Using Interfaces)

1. ***Program Complexity*** *– Using many interfaces may make the program harder to understand.*
2. ***More Implementation Work*** *– The class must implement all the methods defined in the interfaces.*
3. ***Difficult Maintenance*** *– Managing many interfaces in large programs can be challenging.*

Java Program Example

```
interface Engine {
   void startEngine();
}

interface GPS {
   void showLocation();
}

class Vehicle implements Engine, GPS {

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

   public void showLocation() {
      System.out.println("GPS is showing location");
   }
}

public class Main {
   public static void main(String[] args) {

      Vehicle v = new Vehicle();
      v.startEngine();
      v.showLocation();
   }
}
```

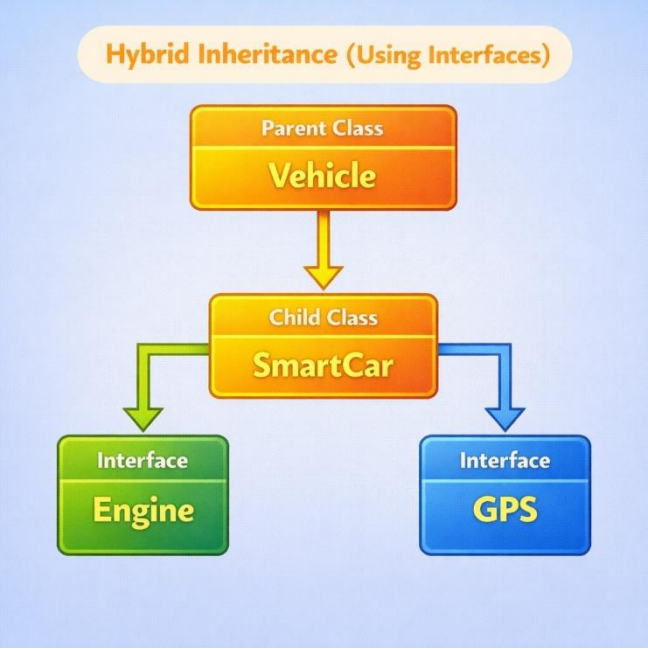
Explanation

In this program, ***Engine*** *and* ***GPS*** *are interfaces that contain method declarations. The* ***Vehicle class implements both interfaces****, which means it inherits the behaviors of both Engine and GPS.*

The Vehicle class provides the implementation for the methods startEngine() and showLocation(). When an object of the Vehicle class is created, it can access the methods of both interfaces.
This demonstrates ***Multiple Inheritance in Java through interfaces****, where* ***one class inherits features from multiple interfaces****.*

5. Hybrid Inheritance in Java (Using Interfaces)

Hybrid inheritance is a type of inheritance that ***combines two or more types of inheritance****, such as single inheritance, multiple inheritance, or hierarchical inheritance. It creates a more complex relationship between classes by combining different inheritance structures in one program.*
In Java, ***hybrid inheritance cannot be implemented using classes alone*** *because Java does not support multiple inheritance with classes. However, hybrid inheritance can be achieved by* ***using interfaces along with classes****. Interfaces allow a class to inherit features from multiple sources without creating ambiguity problems.*
In hybrid inheritance, a class can extend another class and at the same time implement one or more interfaces. This allows the program to combine different functionalities from different sources. As a result, the final class can access the properties and methods of the parent class as well as the methods defined in the interfaces.
The main purpose of hybrid inheritance is to ***increase flexibility and code reuse****. It helps programmers design complex systems by combining different inheritance models in a single structure.*



For example, consider a ***Vehicle*** *class that provides basic vehicle functions. There may also be interfaces such as* ***Engine*** *and* ***GPS*** *that define additional functionalities. A class called* ***SmartCar*** *can extend the Vehicle class and implement both Engine and GPS interfaces. In this way, the SmartCar class inherits features from the Vehicle class and also implements behaviors from the Engine and GPS interfaces.*

Thus, hybrid inheritance in Java allows developers to ***combine different inheritance concepts while maintaining clear and organized program structure****.*

Advantages of Hybrid Inheritance

1. ***High Code Reusability***
   It allows combining features from multiple sources.
2. ***Flexibility in Program Design***
   Different inheritance types can be used together.
3. ***Better Organization***
   Complex systems can be structured more effectively.
4. ***Supports Modular Programming***
   Different functionalities can be separated into interfaces.

Disadvantages of Hybrid Inheritance

1. ***Increased Complexity***
   The structure of the program may become difficult to understand.
2. ***Difficult Maintenance***
   Managing many classes and interfaces can be challenging.
3. ***More Implementation Effort***
   Classes must implement methods from multiple interfaces.

Java Program for Hybrid Inheritance (Using Interfaces)

```
interface Engine {
   void startEngine();
}

interface GPS {
   void showLocation();
}

class Vehicle {
   void move() {
      System.out.println("Vehicle is moving");
   }
}

class SmartCar extends Vehicle implements Engine, GPS {

   public void startEngine() {
      System.out.println("Engine started");
   }

   public void showLocation() {
```

```
        System.out.println("GPS showing location");
    }
}

public class Main {
    public static void main(String[] args) {

        SmartCar s = new SmartCar();

        s.move();          // method from Vehicle class
        s.startEngine();   // method from Engine interface
        s.showLocation();  // method from GPS interface
    }
}
```

Explanation

*In this program, **Vehicle** is a class that provides a basic method called move(). Two interfaces, **Engine** and **GPS**, define additional functionalities.*
*The **SmartCar** class **extends the Vehicle class** and **implements the Engine and GPS interfaces**. Because of this, the SmartCar class can use the method from the Vehicle class and must also provide implementations for the methods declared in the interfaces.*
*This program demonstrates **Hybrid Inheritance**, because it combines **single inheritance (Vehicle → SmartCar)** and **multiple inheritance through interfaces (Engine and GPS)**.*

1.3 Member Access Rules in Java

*In Java, the members of a class include **variables (data members), methods (member functions), and constructors**. The accessibility of these members from other classes is controlled by a set of rules known as **member access rules**. These rules determine **where and how the members of a class can be accessed in a program**.*
*Member access rules are implemented through **access modifiers**. Access modifiers specify the level of visibility of class members and control whether they can be accessed within the same class, within the same package, or from classes in other packages. By using these modifiers, Java ensures that the internal data of a class is properly protected and used in a controlled manner.*
*The concept of member access is closely related to **encapsulation**, which is one of the fundamental principles of object-oriented programming. Encapsulation refers to the process of **hiding the internal details of a class and allowing access only through well-defined methods**. Member access rules help achieve this by restricting direct access to certain variables and methods.*
*Java provides **four types of access modifiers** that define the accessibility of class members.*

1. Private Access Modifier

*The **private access modifier** is the most restrictive access level in Java. When a variable or method is declared as **private**, it can be accessed **only within the same class in which it is***

***declared**. It cannot be accessed directly by other classes, even if those classes belong to the same package.*

*The main purpose of the private access modifier is to **protect the internal data of a class** and prevent unauthorized access from outside the class. This helps maintain **data security and integrity** in a program. By restricting direct access to important variables, the programmer can ensure that the data is modified only in a controlled manner.*

*Private members are commonly used to support the concept of **encapsulation** in object-oriented programming. Encapsulation means **hiding the internal details of a class and allowing access only through specific methods**. In many programs, private variables are accessed using **public getter and setter methods**, which provide a controlled way to read or update the values.*

*For example, in a **BankAccount** class, the variable representing the account balance may be declared as private. This prevents other classes from directly changing the balance. Instead, special methods such as deposit() or withdraw() can be used to modify the balance in a safe way.*

*Thus, the private access modifier plays an important role in **data hiding, security, and proper program design**, making it an essential feature of Java programming.*

Example Program

```
class Student {
   private int marks = 80;

   private void displayMarks() {
      System.out.println("Marks: " + marks);
   }

   public static void main(String[] args) {
      Student s = new Student();
      s.displayMarks();  // accessible inside the same class
   }
}
```

Explanation

*In this example, the variable **marks** and the method **displayMarks()** are declared as private. They can be accessed only within the **Student class**. If another class tries to access them directly, it will result in a compilation error.*

2. Default Access Modifier (Package-Private)

*The **default access modifier** is the access level that is used when **no access modifier is specified** for a class member. In Java, if a variable, method, or class is declared without using public, private, or protected, it automatically gets **default access**. This access level is also known as **package-private access**.*

*Members with default access can be **accessed only by classes that are present in the same package**. Classes that belong to other packages cannot access these members. This helps in*

sharing information between related classes within the same package while still restricting access from outside packages.
Default access is useful when a group of classes in the same package needs to ***work together and share common data or methods****. It keeps the members visible within the package but hides them from unrelated classes in other packages.*
For example, if a class called ***Vehicle*** *has a method called start() without any access modifier, that method can be used by other classes in the same package such as* ***Car*** *or* ***Bike****, but it cannot be accessed by classes in a different package.*
Thus, the default access modifier helps in ***organizing programs into packages and controlling access within the package****, while still protecting the members from external packages.*

Example Program

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

public class Main {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        v.start();   // accessible because it is in the same package
    }
}
```

Explanation
In this program, the ***Vehicle class*** *and the* ***start() method*** *do not have any access modifier. Therefore, they have* ***default (package-private) access****. This means they can be used by other classes in the* ***same package****, but they cannot be accessed from classes in* ***different packages****.*

3. Protected Access Modifier
The ***protected access modifier*** *in Java is used to allow controlled access to class members such as variables and methods. When a member is declared as* ***protected****, it can be accessed* ***within the same class, within the same package, and also by subclasses (child classes)*** *even if those subclasses are located in different packages.*

*The main purpose of the protected access modifier is to **support inheritance while still maintaining some level of protection for class members**. In object-oriented programming, inheritance allows one class to reuse the properties and methods of another class. Protected access makes it possible for subclasses to access important members of the parent class without making those members completely open to all classes.*

*Protected access provides **a balanced level of accessibility**. It is more accessible than **private** and **default access**, but it is more restricted than **public access**. This means that protected members are not available to every class in the program, but they are available to classes that are related through inheritance.*

*When a variable or method is declared as protected, it becomes **part of the inheritance structure**. Subclasses can directly use these members to extend or modify the behavior of the parent class. This helps programmers create reusable and extendable class hierarchies.*

*For example, consider a class called **Vehicle** that has a protected variable called speed. If another class called **Car** extends the Vehicle class, the Car class can access the speed variable directly because it is a subclass. In this way, the Car class can use the features of the Vehicle class and build additional functionality on top of it.*

*Protected members are also accessible to **other classes in the same package**, even if those classes are not subclasses. However, if a class belongs to a **different package and is not a subclass**, it cannot access protected members.*

*Thus, the protected access modifier plays an important role in **supporting inheritance, promoting code reuse, and maintaining controlled access to important class members**. It allows related classes to share functionality while still protecting the internal structure of the class from unrelated classes.*

Example Program

```
class Vehicle {
    protected int speed = 60;
}

class Car extends Vehicle {
    void display() {
        System.out.println("Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.display();
    }
}
```

Explanation

*In this example, the variable **speed** is declared as **protected** in the **Vehicle** class. The **Car** class extends the Vehicle class and can access the speed variable directly. This is possible because protected members are accessible **within the same package and also in subclasses through inheritance**.*

4. Public Access Modifier

*The **public access modifier** provides the **widest level of accessibility** in Java. When a variable, method, or class is declared as **public**, it can be accessed **from anywhere in the program**. This means it can be used by any class, whether the class is in the **same package or in a different package**.*

*The main purpose of the public access modifier is to **make certain members of a class available to other parts of the program**. Public members are usually used when a method or variable needs to be accessed by many classes. For example, methods that display information, perform calculations, or provide services to other classes are often declared as public.*

*Public access is very useful when designing **libraries, frameworks, or reusable classes**. It allows programmers to create methods that can be called by other classes to perform specific tasks. However, public members should be used carefully because making too many variables public can reduce **data security and control**.*

*In object-oriented programming, it is generally recommended to keep **data members private** and provide **public methods** to access or modify them. This approach ensures that the internal data of a class is protected while still allowing controlled interaction with other classes.*

*For example, a class called **Vehicle** may contain a public method called displayDetails() that can be called by any other class to display information about the vehicle. Since the method is public, it can be accessed from anywhere in the program.*

*Thus, the public access modifier plays an important role in **allowing communication between different classes and packages**, making programs more flexible and reusable.*

Example Program

```
class Vehicle {
    public void start() {
        System.out.println("Vehicle is starting");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        v.start();   // accessible from anywhere
    }
}
```

Explanation

*In this program, the method **start()** is declared as **public** in the **Vehicle** class. Because it is public, the method can be accessed by the **Main class** and by any other class in the program, even if it is located in a different package.*

Table: Comparison of Access Modifiers in Java

Access Modifier	***Meaning / Description***	***Same Class***	***Same Package***	***Subclass (Different Package)***	***Other Package***
Private	*Members are accessible only inside the class in which they are declared. It provides the highest level of restriction and supports data hiding.*	*Yes*	*No*	*No*	*No*
Default (Package-Private)	*When no access modifier is specified, the member gets default access. It can be accessed by all classes within the same package but not from other packages.*	*Yes*	*Yes*	*No*	*No*
Protected	*Members can be accessed within the same package and also by subclasses even if they are in different packages. It is mainly used in inheritance.*	*Yes*	*Yes*	*Yes*	*No*
Public	*Members are accessible from anywhere in the program, whether in the same package or different packages. It provides the widest level of access.*	*Yes*	*Yes*	*Yes*	*Yes*

1.4 super Keyword in Java

*In Java, the **super keyword** is a special reference variable that is used to **refer to the immediate parent class (superclass)** of a subclass. It is mainly used in inheritance when a child class needs to access the **properties, methods, or constructors** of its parent class.*
*When a class inherits another class using the **extends keyword**, the subclass automatically obtains the variables and methods of the parent class. However, in some situations the child*

*class needs to **explicitly refer to the members of the parent class**. In such cases, the super keyword is used to clearly indicate that the member belongs to the parent class.*

*The super keyword is especially useful when **both the parent class and the child class contain variables or methods with the same name**. In this situation, the member defined in the child class hides the member of the parent class. By using the super keyword, the child class can access the original member defined in the parent class. This helps remove confusion and ensures that the correct member is used.*

*Another important use of the super keyword is in **method overriding**. When a subclass overrides a method of the parent class, it may still need to use the original implementation provided by the parent class. In such cases, the child class can call the parent class method using the super keyword.*

*The super keyword is also used to **call the constructor of the parent class**. When an object of a subclass is created, the constructor of the parent class is executed first to initialize the inherited members. The statement super() is used inside the child class constructor to explicitly call the constructor of the parent class.*

*Thus, the super keyword plays an important role in Java inheritance. It allows the child class to **access parent class variables, invoke parent class methods, and call parent class constructors**, which helps maintain a clear relationship between classes and promotes **code reuse and better program organization**.*

Uses of super Keyword

1. ***Accessing Parent Class Variables***
   The super keyword is used to access variables of the parent class when the child class defines variables with the same name.
2. ***Calling Parent Class Methods***
   It is used to call methods of the parent class that are overridden in the child class.
3. ***Calling Parent Class Constructor***
   The super() statement is used to call the constructor of the parent class from the constructor of the child class.

Example Program

```
class Vehicle {
   int speed = 50;
}

class Car extends Vehicle {
   int speed = 100;

   void display() {
      System.out.println("Parent class speed: " + super.speed);
      System.out.println("Child class speed: " + speed);
   }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.display();
    }
}
```

Explanation

*In this program, both the **Vehicle class** and the **Car class** contain a variable named speed. The variable in the child class hides the variable in the parent class. By using **super.speed**, the program accesses the speed variable defined in the **Vehicle class**. The variable speed without super refers to the variable of the **Car class**.*

*This example shows how the super keyword helps the child class **access the members of the parent class clearly and avoid ambiguity**.*

1.5 final Keyword in Java

*In Java, the **final keyword** is used to **restrict modification**. It indicates that once a variable, method, or class is declared as final, it cannot be changed or extended further. The final keyword is mainly used to make the program **more secure, stable, and reliable** by preventing unwanted changes.*

In object-oriented programming, there are situations where certain elements of a program should remain fixed. For example, some values should not change, some methods should not be overridden, and some classes should not be inherited. The final keyword helps achieve this control by restricting further modification.

*The final keyword can be applied to **variables, methods, and classes**, and its meaning depends on where it is used.*

---

1. Final Variable

*When a variable is declared as **final**, its value cannot be changed once it is assigned. This means the variable becomes a **constant**. After the value is assigned, it cannot be modified again in the program.*

*Final variables are commonly used when a value must remain constant throughout the program. For example, mathematical values like **PI** or fixed configuration values can be declared as final.*

Example

```
class Example {
    final int number = 10;

    void display() {
        System.out.println("Number: " + number);
    }
```

```
    public static void main(String[] args) {
        Example obj = new Example();
        obj.display();
    }
}
```

*In this example, the variable number is declared as **final**, so its value cannot be changed after initialization.*

---

2. Final Method

*When a method is declared as **final**, it cannot be **overridden by a subclass**. This means that the method defined in the parent class must be used exactly as it is, and child classes cannot change its implementation.*

*Final methods are useful when the programmer wants to **protect the original behavior of a method** and prevent subclasses from modifying it.*

Example

```
class Vehicle {
    final void start() {
        System.out.println("Vehicle is starting");
    }
}

class Car extends Vehicle {
    // start() method cannot be overridden here
}
```

In this example, the method start() is declared as final, so the Car class cannot override it.

---

3. Final Class

*When a class is declared as **final**, it cannot be **inherited by other classes**. This means no subclass can be created from that class.*

*Final classes are used when the programmer wants to **prevent inheritance and keep the class implementation unchanged**.*

Example

```
final class Vehicle {
    void display() {
        System.out.println("This is a final class");
    }
}

// class Car extends Vehicle { }  // This will cause an error
```

In this example, the class Vehicle is declared as final, so it cannot be extended by another class.

2. Abstract Classes and Polymorphism

2.1 Abstract Classes

*In Java, an **abstract class** is a class that is declared using the **abstract keyword**. It is used to represent a general concept and serves as a **base class for other classes**. An abstract class cannot be instantiated directly, which means **objects cannot be created from an abstract class**. Instead, it is designed to be inherited by other classes that provide the implementation for its abstract methods.*

*An abstract class may contain **both abstract methods and non-abstract methods**. Abstract methods are methods that do not have a body, while non-abstract methods contain a complete implementation. The subclasses that inherit the abstract class must provide the implementation for all abstract methods.*

*The main purpose of an abstract class is to **provide a common structure or blueprint for a group of related classes**. When several classes share common characteristics, these features can be defined in an abstract class. The subclasses can then inherit the common features and also implement their own specific behavior.*

*Abstract classes help in achieving **abstraction**, which means hiding unnecessary details and focusing only on the essential features. By using abstract classes, programmers can create a **well-organized class hierarchy** and avoid repeating the same code in multiple classes.*

*For example, consider a class called **Vehicle**. A vehicle may have a function called start(), but the way a vehicle starts may differ depending on whether it is a **Car, Bike, or Bus**. In this situation, the **Vehicle** class can be declared as an abstract class, and the subclasses such as Car and Bike will provide the specific implementation for the start() method.*

2.2 Abstract Methods

*An **abstract method** is a method that is declared without an implementation. It contains only the **method declaration and does not include a method body**. Abstract methods are declared using the **abstract keyword** and are usually defined inside an abstract class.*

*The purpose of an abstract method is to **define a method that must be implemented by the subclasses**. When a class inherits an abstract class, it must provide the implementation for all the abstract methods of that class. If the subclass does not implement the abstract methods, then the subclass must also be declared as abstract.*

*Abstract methods help ensure that all subclasses **follow a common structure while still allowing different implementations**. This is useful when different classes need to perform the same operation but in different ways.*

*For example, if an abstract class **Vehicle** contains an abstract method start(), then all subclasses such as **Car, Bike, and Bus** must implement the start() method according to their own functionality. This ensures that every type of vehicle provides its own version of the starting process.*

*Thus, abstract methods play an important role in **achieving abstraction and maintaining consistency among related classes** in Java programs.*

Program to Demonstrate Abstract Class and Abstract Method

```
abstract class Vehicle {

    // Abstract method
    abstract void start();

    // Normal method
    void display() {
        System.out.println("This is a vehicle");
    }
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car starts with a key or button");
    }
}

class Bike extends Vehicle {
    void start() {
        System.out.println("Bike starts with a kick or self-start");
    }
}

public class Main {
    public static void main(String[] args) {

        Vehicle v1 = new Car();
        Vehicle v2 = new Bike();

        v1.display();
        v1.start();

        v2.display();
        v2.start();
    }
}
```

Explanation

*In this program, **Vehicle** is declared as an **abstract class**. It contains an **abstract method** called start() and a normal method called display().*

*The classes **Car** and **Bike** extend the **Vehicle** class and provide the implementation for the abstract method start(). Since the method start() has no implementation in the abstract class, it must be implemented in the subclasses.*

*Objects cannot be created for the **Vehicle** class because it is abstract. Instead, objects are created for the **Car** and **Bike** classes. This program demonstrates how **abstract classes provide a common structure, while abstract methods allow subclasses to define their own specific behavior**.*

Difference Between Abstract Class and Abstract Method

Aspect	***Abstract Class***	***Abstract Method***
Definition	*An abstract class is a class declared using the abstract keyword that cannot be instantiated and serves as a base class for other classes.*	*An abstract method is a method declared using the abstract keyword without any implementation (no method body).*
Purpose	*It is used to provide a **common structure or blueprint** for a group of related classes.*	*It is used to define a **method that must be implemented by subclasses**.*
Implementation	*It can contain **both abstract methods and normal methods with implementation**.*	*It contains **only the method declaration and no implementation**.*
Object Creation	*Objects **cannot be created directly** from an abstract class.*	*Abstract methods do not involve object creation because they are only declarations.*
Usage in Program	*It is used when a class represents a **general concept that should not be instantiated directly**.*	*It is used when a method must be **implemented differently by each subclass**.*
Role in Inheritance	*Subclasses **inherit the abstract class** and use its properties and methods.*	*Subclasses **must override the abstract method** and provide its implementation.*
Example	*abstract class Vehicle { }*	*abstract void start();*

Java Program

```
abstract class Payment {

    // Abstract method
    abstract void processPayment();

    // Normal method
    void display() {
        System.out.println("Processing payment...");
    }
}

class CreditCardPayment extends Payment {
    void processPayment() {
        System.out.println("Payment done using Credit Card");
```

```
    }
}

class DebitCardPayment extends Payment {
    void processPayment() {
        System.out.println("Payment done using Debit Card");
    }
}

class UpiPayment extends Payment {
    void processPayment() {
        System.out.println("Payment done using UPI");
    }
}

public class Main {
    public static void main(String[] args) {

        Payment p1 = new CreditCardPayment();
        Payment p2 = new DebitCardPayment();
        Payment p3 = new UpiPayment();

        p1.display();
        p1.processPayment();

        p2.display();
        p2.processPayment();

        p3.display();
        p3.processPayment();
    }
}
```

Explanation

In this program, ***Payment*** *is an* ***abstract class*** *that represents a general payment system. It contains an abstract method called* ***processPayment()****, which does not have an implementation.*

The classes ***CreditCardPayment****,* ***DebitCardPayment****, and* ***UpiPayment*** *extend the Payment class and provide their own implementation of the processPayment() method.*

This example shows that the ***abstract class provides the common structure****, while the* ***abstract method ensures that each subclass defines its own specific behavior****.*

2.3 Polymorphism

Polymorphism is one of the most powerful and essential principles of ***Object-Oriented Programming (OOP)*** *in Java. The term* ***polymorphism*** *originates from two Greek words:* ***"poly" meaning many*** *and* ***"morph" meaning forms****. Therefore, polymorphism literally means* ***"many forms."***

In the context of programming, polymorphism refers to the ***ability of a single method, interface, or object to represent multiple behaviors depending on the situation in which it is used****. In other words, the same operation may behave differently when applied to different objects or when used with different types of data.*

Polymorphism is important because it allows programmers to ***write flexible and reusable code****. Instead of creating separate methods for each type of operation, a programmer can use* ***one method name that performs different tasks depending on the context****. This approach reduces redundancy in code and improves the readability and maintainability of programs.*

Another important aspect of polymorphism is that it enables objects belonging to different classes to ***respond differently to the same method call****. This means that even though multiple classes may use the same method name, the actual behavior of that method depends on the* ***type of object that invokes it****.*

Polymorphism also plays a major role in ***inheritance and abstraction****. When classes share a common parent class or interface, polymorphism allows the program to treat objects of different subclasses in a* ***uniform manner****, while still allowing each subclass to implement its own specific behavior.*

For example, consider a general class called ***Shape*** *that defines a method called draw(). Different shapes such as* ***Circle, Rectangle, and Triangle*** *may implement the draw() method differently. Even though the method name remains the same, the* ***actual drawing behavior changes depending on the shape object****.*

In Java, polymorphism mainly occurs in ***two different forms****, depending on* ***when the method call is resolved****:*

1. ***Compile-time Polymorphism (Method Overloading)***
2. ***Run-time Polymorphism (Method Overriding)***

These two forms allow Java programs to achieve flexibility either ***during compilation or during program execution****.*

2.3.1 Method Overriding (Run-time Polymorphism)

Method overriding is a mechanism in Java where a ***subclass provides its own implementation of a method that is already defined in its parent class****. The method in the child class must have the* ***same method name, same parameters, and same return type*** *as the method in the parent class.*

Method overriding supports ***runtime polymorphism****, meaning that the decision about which method should be executed is made* ***during program execution rather than during compilation****.*

In runtime polymorphism, a ***parent class reference variable can refer to objects of different subclasses****. When a method is called using this reference variable, the* ***actual method that is executed depends on the object type that the reference variable points to at runtime****.*

*This concept is also known as **dynamic method dispatch**, because the method call is dynamically resolved during execution.*

*The main advantage of method overriding is that it allows subclasses to **modify or specialize the behavior of a parent class method**. This means that a general method defined in the parent class can be adapted to meet the specific requirements of each subclass.*

*For example, consider a parent class called **Vehicle** with a method called start(). Different types of vehicles may start in different ways. A **Car** may start with a key or button, while a **Bike** may start with a kick or self-start mechanism. Each subclass can override the start() method and provide its own implementation.*

Example Program

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

class Car extends Vehicle {
    void start() {
        System.out.println("Car starts using a key");
    }
}

class Bike extends Vehicle {
    void start() {
        System.out.println("Bike starts using a kick");
    }
}

public class Main {
    public static void main(String[] args) {

        Vehicle v1 = new Car();
        Vehicle v2 = new Bike();

        v1.start();
        v2.start();
    }
}
```

*In this program, the parent class **Vehicle** defines the method start(). The subclasses **Car** and **Bike** override this method and provide their own implementations.*

*Even though the reference variable type is **Vehicle**, the method that gets executed depends on the **actual object type** (Car or Bike). This decision is made **during runtime**, which demonstrates runtime polymorphism.*

2.3.2 Method Overloading (Compile-time Polymorphism)

Method overloading is a feature in Java that allows ***multiple methods in the same class to have the same name but different parameter lists****. The difference between overloaded methods can occur in:*

- *The* ***number of parameters***
- *The* ***type of parameters***
- *The* ***order of parameters***

Method overloading represents ***compile-time polymorphism****, which means the decision about which method should be executed is made* ***during compilation*** *rather than during runtime.*

When the compiler encounters a method call, it examines the ***number and type of arguments provided*** *and determines which version of the method should be invoked. Because this decision is made before the program runs, it is called compile-time polymorphism or* ***static polymorphism****.*

The main purpose of method overloading is to allow programmers to ***use the same method name for similar operations involving different types or numbers of inputs****. This improves program readability because the programmer does not need to create separate method names for each operation.*

For example, consider a method called add() used for addition. The same method name can be used to add ***two integers, three integers, or two decimal numbers****.*

Example Program

```
class Calculator {

    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }

    double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {

        Calculator obj = new Calculator();

        System.out.println(obj.add(5, 3));
        System.out.println(obj.add(5, 3, 2));
        System.out.println(obj.add(2.5, 3.5));
```

```
    }
}
```

*In this program, the method **add()** is defined multiple times with different parameter lists. Even though the method name is the same, each method performs a different operation depending on the parameters.*

*When the method is called, the compiler determines which version of the method should be executed based on the **arguments provided in the method call**. Because the method selection happens during compilation, it is called **compile-time polymorphism**.*

Comparison Between Compile-Time Polymorphism and Run-Time Polymorphism

Aspect	***Compile-Time Polymorphism***	***Run-Time Polymorphism***
Definition	*Compile-time polymorphism occurs when the method call is resolved during the compilation of the program. The compiler determines which method should be executed based on the method parameters.*	*Run-time polymorphism occurs when the method call is resolved during program execution. The method that is executed depends on the actual object type at runtime.*
Other Name	*It is also called **Static Polymorphism** because the method call is decided before the program runs.*	*It is also called **Dynamic Polymorphism** because the method execution is decided while the program is running.*
Resolution Time	*The decision about which method should be executed is made **during compilation** by the compiler.*	*The decision about which method should be executed is made **during runtime** by the Java Virtual Machine (JVM).*
Achieved By	*It is achieved through **Method Overloading**, where multiple methods have the same name but different parameter lists.*	*It is achieved through **Method Overriding**, where a subclass provides its own implementation of a method defined in the parent class.*
Inheritance Requirement	*Inheritance is **not required**. Method overloading can occur within a single class.*	*Inheritance is **required** because method overriding occurs between parent and child classes.*
Method Signature	*Methods must have the **same name but different parameters** (number, type, or order of parameters).*	*Methods must have the **same name, same parameters, and same return type** as the parent class method.*
Binding	*Uses **Early Binding (Static Binding)** because the method is determined during compilation.*	*Uses **Late Binding (Dynamic Binding)** because the method is determined during runtime.*
Performance	*Generally **faster** because the method call is resolved before execution.*	*Slightly **slower** because the method is determined during runtime using dynamic binding.*

Flexibility	*Provides flexibility in using the same method name with different parameter lists within a class.*	*Provides flexibility by allowing subclasses to define specific behavior for inherited methods.*
Example	*A class having multiple add() methods with different parameters.*	*A subclass overriding a start() method of a parent class such as Vehicle.*

3. Interfaces

*In Java, an **interface** is a special type of reference type that is used to define a **set of abstract methods that a class must implement**. An interface acts as a **contract or specification** that defines the behavior a class must follow. It specifies **what operations should be performed**, but it does not describe **how those operations should be implemented**.*

*Interfaces are declared using the **interface keyword**. Unlike normal classes, interfaces generally do not contain complete method implementations. Instead, they contain **method declarations**, and the responsibility of implementing these methods is given to the classes that implement the interface.*

*The concept of interfaces is closely related to **abstraction**. Abstraction is the process of hiding implementation details and showing only the essential features of an object. Interfaces help achieve **complete abstraction** because they focus only on defining behavior without providing the internal implementation.*

*Another important advantage of interfaces is that they support **multiple inheritance in Java**. Java does not allow a class to inherit from multiple classes because it may lead to ambiguity. However, a class can implement multiple interfaces, which allows it to inherit behaviors from different sources.*

*Interfaces are widely used in large software systems because they help create **loosely coupled and highly flexible programs**. By using interfaces, programmers can design systems where different classes can interact through a common interface without depending on the internal implementation of each class.*

*For example, consider a **Vehicle interface** that declares methods such as start() and stop(). Different classes such as **Car, Bike, and Bus** can implement this interface and provide their own implementation of these methods. In this way, all vehicles follow the same interface but may behave differently.*

*Thus, interfaces provide a **standard structure that ensures consistency, reusability, and flexibility in object-oriented programs**.*

3.1 Defining an Interface

*Defining an interface means creating an interface that contains **method declarations and constant variables** that must be implemented by classes. An interface is declared using the **interface keyword**, followed by the interface name.*

*Inside an interface, methods are declared **without a body**, because the implementation is provided later by the implementing classes. These methods are automatically considered **public and abstract**, even if these keywords are not explicitly written.*

Similarly, all variables declared in an interface are automatically ***public, static, and final****, meaning they behave like constants whose values cannot be changed.*

The purpose of defining an interface is to ***describe a common behavior that multiple classes can share****. It provides a blueprint that ensures all implementing classes follow the same structure.*

Example: Defining an Interface

```
interface Vehicle {
    void start();
    void stop();
}
```

Explanation

In this example, the ***Vehicle interface*** *defines two methods: start() and stop(). These methods do not contain any implementation. They only define the behavior that must be implemented by any class that uses this interface.*

3.2 Implementing Interfaces

Implementing an interface means creating a class that ***provides the actual implementation for all the methods declared in the interface****. A class implements an interface using the* ***implements keyword****.*

When a class implements an interface, it must define the body of every method declared in the interface. If the class does not provide implementations for all the methods, the class must be declared as ***abstract****.*

The process of implementing interfaces allows different classes to ***share a common set of behaviors while maintaining their own unique functionality****.*

One of the key advantages of implementing interfaces is that ***a single interface can be implemented by multiple classes****, and each class can provide its own specific implementation.*

Example: Implementing an Interface

```
interface Vehicle {
    void start();
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car starts with a key");
    }
}

class Bike implements Vehicle {
    public void start() {
        System.out.println("Bike starts with a kick");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        Bike b = new Bike();

        c.start();
        b.start();
    }
}
```

Explanation

*In this example, the **Vehicle interface** declares the method start(). The classes **Car** and **Bike** implement the Vehicle interface and provide their own implementation of the start() method. Although the method name is the same, the behavior differs depending on the class.*

3.3 Extending Interface

*Just like classes, interfaces can also inherit from other interfaces. This is done using the **extends keyword**. When one interface extends another interface, it **inherits all the method declarations of the parent interface**.*

*The child interface can also declare additional methods. Any class that implements the child interface must provide implementations for **all the methods declared in both the parent and child interfaces**.*

*Extending interfaces helps in creating a **hierarchical structure of interfaces**, which allows programmers to design complex systems in an organized way.*

*Unlike classes, an interface can **extend multiple interfaces at the same time**, which further supports multiple inheritance in Java.*

Example: Extending Interface

```
interface Vehicle {
    void start();
}

interface ElectricVehicle extends Vehicle {
    void charge();
}

class ElectricCar implements ElectricVehicle {

    public void start() {
        System.out.println("Electric car starts with a button");
    }

    public void charge() {
        System.out.println("Electric car is charging");
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        ElectricCar e = new ElectricCar();
        e.start();
        e.charge();
    }
}
```

Explanation

*In this program, the **ElectricVehicle interface extends the Vehicle interface**, which means it inherits the start() method. It also declares an additional method called charge().*
*The class **ElectricCar** implements the ElectricVehicle interface and therefore provides implementations for both start() and charge() methods.*

Difference Between Abstract Class and Interface
Comparison Table

Aspect	***Abstract Class***	***Interface***
Definition	*An abstract class is a class declared with the abstract keyword that may contain both abstract methods and concrete methods.*	*An interface is a reference type that defines a set of abstract methods that must be implemented by classes.*
Purpose	*It is used to provide a **base class with common functionality** for related classes.*	*It is used to define a **contract that specifies behavior** that classes must implement.*
Methods	*It can contain **abstract methods and normal methods with implementation**.*	*Traditionally contains **only abstract methods** (method declarations without implementation).*
Variables	*It can contain **instance variables, static variables, and constants**.*	*Variables are **public, static, and final by default**.*
Constructors	*Abstract classes **can have constructors**.*	*Interfaces **do not have constructors**.*
Access Modifiers	*Methods and variables can have **private, protected, or public** access modifiers.*	*Methods are **public by default**, and variables are **public, static, and final**.*
Inheritance	*A class can **extend only one abstract class**.*	*A class can **implement multiple interfaces**.*
Keyword Used	*The **extends keyword** is used to inherit an abstract class.*	*The **implements keyword** is used to implement an interface.*
Implementation	*Abstract classes can provide **partial implementation**.*	*Interfaces generally provide **only method declarations**.*

Usage	*Used when classes share **common properties and behavior**.*	*Used when different classes must follow **a common set of rules or capabilities**.*

3.4 Constructor Overloading

*In Java, a **constructor** is a special member function of a class that is used to **initialize the objects of that class**. A constructor is automatically invoked when an object is created. It has the **same name as the class** and does not have any return type. The main purpose of a constructor is to assign initial values to the data members of the object at the time of object creation.*

*Sometimes a class may require **different ways to initialize its objects** depending on the information available at the time of object creation. In such situations, Java provides the concept of **constructor overloading**.*

*Constructor overloading refers to the process of **defining multiple constructors in the same class with the same name but different parameter lists**. These constructors differ in terms of the **number of parameters, type of parameters, or order of parameters**. Each constructor performs a different type of initialization depending on the arguments passed while creating the object.*

*The main advantage of constructor overloading is that it allows a class to **initialize objects in different ways without using different constructor names**. This improves the flexibility and usability of the class because the programmer can choose the appropriate constructor depending on the available data.*

*When an object is created, the **Java compiler determines which constructor should be executed based on the arguments provided during object creation**. This selection is done at **compile time**, making constructor overloading similar to **compile-time polymorphism**.*

*Constructor overloading also improves the **readability and maintainability of programs**. Instead of creating separate initialization methods, multiple constructors can be used to perform different types of object initialization within the same class. This helps in writing more organized and efficient code.*

*For example, consider a **Student class**. Sometimes a student object may be created without providing any details, while in other cases the student's **ID and name** may be provided during object creation. In such cases, constructor overloading allows the programmer to define **multiple constructors** so that objects can be created in different ways.*

*Thus, constructor overloading is an important concept in Java that provides **flexibility in object creation, supports different initialization requirements, and enhances the overall design of object-oriented programs**.*

Example Program

```
class Student {

    int id;
    String name;
```

```
    // Default Constructor
    Student() {
        id = 0;
        name = "Unknown";
    }

    // Parameterized Constructor
    Student(int i, String n) {
        id = i;
        name = n;
    }

    void display() {
        System.out.println("ID: " + id + " Name: " + name);
    }

    public static void main(String[] args) {

        Student s1 = new Student();
        Student s2 = new Student(101, "Rahul");

        s1.display();
        s2.display();
    }
}
```

Explanation

*In this program, the **Student class contains two constructors**. The first constructor is a **default constructor** that initializes the object with default values, while the second constructor is a **parameterized constructor** that initializes the object with specific values provided during object creation.*

*When the object s1 is created without arguments, the **default constructor** is called. When the object s2 is created with arguments, the **parameterized constructor** is executed. The compiler determines which constructor to invoke based on the parameters passed at the time of object creation.*

Advantages of Constructor Overloading

1. *It allows **multiple ways to initialize objects**.*
2. *It improves **code readability and flexibility**.*
3. *It helps in **proper object initialization based on different conditions**.*
4. *It supports **compile-time polymorphism** in Java.*

4. Packages

*In Java, a **package** is a namespace that is used to **organize related classes and interfaces into a single unit**. Packages help group classes that perform similar tasks so that they can be easily managed and reused in different programs. By using packages, programmers can avoid naming conflicts and maintain a well-structured program.*

*In large software projects, many classes may have the same name. If all classes are placed in a single location, it becomes difficult to manage them. Packages solve this problem by **organizing classes into separate groups**, which improves the readability and maintainability of programs.*

*Packages also provide **access protection**, which means they help control the visibility of classes and their members. By using packages, programmers can specify which classes can be accessed from other packages and which should remain restricted.*

Java provides two types of packages:

1. ***Inbuilt Packages***
2. ***User-Defined Packages***

4.1 Inbuilt Packages

*In Java, **inbuilt packages** (also known as **predefined or standard packages**) are packages that are **already provided by the Java Development Kit (JDK)**. These packages contain a large collection of **predefined classes and interfaces** that help programmers perform common programming tasks easily. Instead of writing code for basic operations from scratch, developers can use these ready-made classes and methods available in inbuilt packages. This makes programming faster, more efficient, and easier to manage.*

*Inbuilt packages are an important part of the **Java Standard Library** and provide support for various functions such as **input-output operations, mathematical calculations, data structures, networking, file handling, and graphical user interface development**. By using these packages, programmers can focus more on the logic of the program rather than implementing basic utilities. Some packages are automatically available in every Java program, while others must be imported using the **import keyword** before they can be used.*

Commonly Used Inbuilt Packages in Java

1. ***java.lang***
   This package contains fundamental classes such as String, Math, System, and Object. It is automatically imported by the Java compiler.
2. ***java.util***
   This package provides utility classes such as Scanner, ArrayList, HashMap, Date, and Random, which are commonly used for data handling and collections.
3. ***java.io***
   *This package contains classes used for **input and output operations**, including file handling such as File, FileReader, and FileWriter.*
4. ***java.awt***
   *This package is used for creating **Graphical User Interface (GUI)** components such as buttons, windows, labels, and text fields.*

5. ***java.net***
   *This package provides classes for **networking operations**, including communication between computers through the internet.*

*Thus, inbuilt packages play an important role in Java programming because they provide **ready-made classes and methods that simplify development, reduce coding effort, and improve program efficiency**.*

4.2 User-Defined Packages

*In Java, **user-defined packages** are packages that are **created by programmers to organize their own classes and interfaces into logical groups**. While Java provides many predefined packages, programmers often need to create their own packages when developing large or complex applications.*

*The primary purpose of user-defined packages is to **organize classes into meaningful groups so that programs become easier to understand, maintain, and reuse**. When many classes are created for a project, placing all of them in a single location can make the program difficult to manage. By grouping related classes into packages, programmers can create a **structured hierarchy of classes**, which improves the readability and organization of the program.*

*User-defined packages also help **prevent naming conflicts**. In large software systems, different programmers may create classes with the same name. Without packages, this could cause errors or confusion. By placing classes in different packages, Java distinguishes them using the **package name along with the class name**, which ensures that each class has a unique identity.*

*Another important benefit of user-defined packages is **code reuse and modular programming**. Classes stored in a package can be reused in other programs simply by importing the package. This allows programmers to divide a large program into smaller modules, each stored in its own package, which makes development easier and more efficient.*

*User-defined packages also provide **access protection**. By using access modifiers such as public, protected, and default access, programmers can control the visibility of classes and their members outside the package. This helps maintain the security and integrity of the program.*

In Java, working with user-defined packages generally involves three main steps:

1. ***Defining the package***
2. ***Creating the package directory***
3. ***Accessing the package in another program***

4.2.1 Defining a Package

*Defining a package means **declaring the name of the package in which a class belongs**. This is done using the **package keyword**.*

*The package statement must appear **at the very beginning of the program**, before any class or interface definitions. This statement informs the Java compiler that the class should belong to the specified package.*

*By defining a package, the programmer creates a **namespace** for the class, which helps organize related classes together.*

Example

```
package mypackage;

public class Test {
    public void display() {
        System.out.println("This is a user-defined package example");
    }
}
```

Explanation

*In this example, the class **Test** is declared inside the package **mypackage**. This means that the compiled class file will be stored inside a directory named **mypackage**, and the class can later be accessed using the package name.*

4.2.2 Creating a Package

*After defining a package, the next step is **creating the package structure in the file system**. This is done during the compilation of the program.*

*Java provides the **-d option with the javac compiler** to automatically create the directory structure that corresponds to the package name. This option instructs the compiler to place the compiled class files into the appropriate directories.*

Example Command

```
javac -d . Test.java
```

Explanation

- *The -d option tells the compiler to create the required package directories.*
- *The . symbol represents the **current directory**.*
- *After compilation, a directory named **mypackage** will be created automatically.*
- *The compiled class file **Test.class** will be stored inside this directory.*

*This ensures that the **directory structure matches the package structure**, which is required for proper package management in Java.*

4.2.3 Accessing a Package

*Accessing a package means **using the classes that are stored in a package within another program**. There are two main ways to access classes from a package:*

1. *Using the **fully qualified class name***
2. *Using the **import statement***

*When the fully qualified class name is used, the class is referred to by writing the **package name followed by the class name**.*

Example Using Fully Qualified Name

```
class Demo {
    public static void main(String[] args) {
        mypackage.Test obj = new mypackage.Test();
        obj.display();
    }
}
```

Explanation

*In this example, the **Test** class from the package **mypackage** is accessed by writing the complete name mypackage.Test. This tells the Java compiler exactly where the class is located.*

4.3 Importing Packages

*Importing a package allows a program to **use classes from another package without writing the full package name every time**. The import keyword is used for this purpose.*
By importing a package, programmers can simplify their code and improve readability.
There are two main ways to import packages:

---

1. Importing a Specific Class

A single class from a package can be imported by specifying the full package path followed by the class name.

Example

import java.util.Scanner;

Explanation

*This statement imports the **Scanner class** from the java.util package. After importing it, the program can directly use the Scanner class without writing java.util.Scanner each time.*

---

2. Importing All Classes of a Package

*It is also possible to import all classes from a package using the * symbol.*

Example

import java.util.;*

Explanation

*This statement imports **all classes available in the java.util package**, such as Scanner, ArrayList, HashMap, Random, and Date.*

Advantages of Packages

*Packages play an important role in Java programming by helping programmers **organize classes and interfaces into logical groups**. They provide several advantages that make programs easier to develop, understand, and maintain.*

1. ***Better Organization of Classes***
   Packages help in organizing related classes and interfaces into a single unit. This makes the program structure more systematic and easier to understand, especially when dealing with large applications that contain many classes.
2. ***Avoidance of Naming Conflicts***
   In large projects, different programmers may create classes with the same name. Packages help avoid naming conflicts by placing classes in different namespaces. As a result, classes with the same name can exist in different packages without causing errors.
3. ***Code Reusability***
   Packages promote code reuse because classes stored in a package can be used in

other programs by importing the package. This reduces duplication of code and improves the efficiency of program development.

4. ***Improved Program Maintainability***
   By organizing classes into packages, programmers can easily locate and modify specific parts of the program. This makes it easier to maintain and update the program when changes are required.
5. ***Access Protection and Security***
   Packages provide access protection by working together with access modifiers such as public, protected, and default access. This allows programmers to control the visibility of classes and their members, ensuring that only the required components are accessible outside the package.
6. ***Support for Modular Programming***
   Packages support modular programming by allowing large programs to be divided into smaller modules. Each module can be developed and tested independently, which improves the overall efficiency and reliability of software development.

*Thus, packages help create **well-structured, reusable, and maintainable Java programs**, making them an essential feature in large-scale software development.*

5. Exception Handling in Java

*Exception Handling in Java is a powerful mechanism used to manage errors that occur during the execution of a program. In real-world programming, it is common for unexpected situations to arise while a program is running. For example, a user may enter invalid data, a program may try to divide a number by zero, or it may attempt to access a file that does not exist. Such unexpected events are known as **exceptions**.*

*An **exception** is an abnormal condition that disrupts the normal flow of a program during its execution. When an exception occurs, the program may terminate suddenly if it is not handled properly. This abrupt termination may prevent the remaining statements in the program from executing and may cause loss of important data or incomplete operations.*

*To overcome this problem, Java provides a structured mechanism called **Exception Handling**. It allows programmers to detect runtime errors and handle them gracefully so that the program can continue executing normally or terminate in a controlled manner. Instead of allowing the program to crash, exception handling enables the program to respond to errors with meaningful messages and corrective actions.*

*Exception handling improves the **reliability, maintainability, and robustness** of a program. It also separates the code that deals with errors from the normal program logic, which makes the program easier to read and understand.*

In Java, exception handling is implemented using five important keywords:

1. ***try block***
2. ***catch block***
3. ***finally block***
4. ***throw keyword***
5. ***throws keyword***

Each component plays a specific role in the exception handling process.

5.1.1.try Block

*The **try block** is a fundamental component of Java's exception handling mechanism. It is used to enclose the section of code that may potentially generate an exception during program execution. In many programs, certain operations such as arithmetic calculations, file handling, database access, or array manipulation may cause runtime errors. To handle such situations safely, these statements are placed inside a try block.*

When the Java Virtual Machine executes the statements inside the try block, it continuously monitors them for any abnormal conditions. If no exception occurs, the program continues executing the remaining statements normally. However, if an exception arises within the try block, the JVM immediately stops executing the remaining statements in that block and searches for a suitable catch block to handle the exception.

*The try block therefore acts as a **protective boundary around risky code** that might cause runtime errors.*

Important Points about try Block

- *The try block contains the statements that may generate an exception.*
- *It must be followed by at least one **catch block** or a **finally block**.*
- *Multiple catch blocks can follow a single try block.*
- *If an exception occurs, the remaining statements inside the try block are skipped.*
- *Control is transferred to the appropriate catch block.*

Syntax

```
try
{
   // statements that may cause an exception
}
```

Example

```
class TryExample
{
   public static void main(String args[])
   {
      try
      {
         int a = 20;
         int b = 0;
         int c = a / b;
         System.out.println(c);
      }
   }
}
```

In this example, the statement a / b may produce an exception because division by zero is not allowed.

5.1.2 Catch Block

*The **catch block** is used to handle the exceptions that occur within the try block. When an exception is generated in the try block, the Java runtime system searches for a catch block that matches the type of exception.*
If a matching catch block is found, control is transferred to that block and the statements inside it are executed. The catch block prevents the program from terminating abruptly and allows the programmer to provide meaningful error messages or corrective actions.
*The catch block receives an **exception object** as a parameter. This object contains details about the exception such as the error message and the location where the error occurred.*

Important Points about catch Block

- *A catch block handles the exception thrown in the try block.*
- *It must immediately follow the try block.*
- *Multiple catch blocks can be used to handle different types of exceptions.*
- *Each catch block specifies the type of exception it can handle.*
- *If no matching catch block is found, the program terminates.*

Syntax

```
try
{
   // risky code
}
catch(ExceptionType e)
{
   // handling code
}
```

Example

```
class CatchExample
{
   public static void main(String args[])
   {
      try
      {
         int a = 10;
         int b = 0;
         int c = a / b;
      }
      catch(ArithmeticException e)
      {
         System.out.println("Division by zero is not allowed.");
      }
   }
}
```

*Here, the catch block handles the **ArithmeticException** that occurs due to division by zero.*

5.1.3 throw Keyword

*The **throw keyword** is used to explicitly create and generate an exception within a program. Normally, exceptions are generated automatically by the Java runtime system when an abnormal condition occurs. However, sometimes a programmer may want to generate an exception manually based on certain conditions in the program logic.*
In such situations, the throw keyword is used to throw an exception object.
The throw statement is followed by an object of an exception class.

Important Points about throw Keyword

- *It is used to manually generate an exception.*
- *It is followed by an object of the exception class.*
- *It transfers control to the nearest catch block.*
- *It is commonly used for validating program conditions.*

Syntax

```
throw new ExceptionType("Error message");
```

Example

```
class ThrowExample
{
   public static void main(String args[])
   {
      int age = 16;

      if(age < 18)
      {
         throw new ArithmeticException("You are not eligible to vote");
      }

      System.out.println("You can vote.");
   }
}
```

In this example, the program manually throws an exception when the age is less than 18.

5.1.4 throws Keyword

*The **throws keyword** is used in the method declaration to specify that a method may generate one or more exceptions during its execution. Instead of handling the exception inside the method, the method can declare it using the throws keyword so that the responsibility of handling the exception is transferred to the calling method.*
*This concept is known as **exception propagation**.*
The throws keyword is commonly used when a method performs operations that may generate checked exceptions.

Important Points about throws Keyword

- *It is used in the **method declaration**.*
- *It declares that a method may produce an exception.*
- *It passes the responsibility of handling the exception to the calling method.*

- *It is mainly used with **checked exceptions**.*

Syntax

```
returnType methodName() throws ExceptionType
{
   // method code
}
```

Example

```
import java.io.*;

class ThrowsExample
{
   static void readFile() throws IOException
   {
      FileReader file = new FileReader("data.txt");
   }

   public static void main(String args[])
   {
      try
      {
         readFile();
      }
      catch(IOException e)
      {
         System.out.println("File not found.");
      }
   }
}
```

*In this example, the method readFile() declares that it may generate an **IOException**, and the main method handles the exception.*

5.1.5 finally Block

*The **finally block** is used to execute important statements that must run regardless of whether an exception occurs or not. It is usually used for performing cleanup operations such as closing files, releasing system resources, closing database connections, or freeing memory. Even if an exception occurs and is handled in the catch block, the finally block will always execute before the program continues.*

This ensures that important resources are properly released and the program remains stable.

Important Points about finally Block

- *It always executes after the try and catch blocks.*
- *It executes whether an exception occurs or not.*
- *It is commonly used for cleanup operations.*
- *It is optional but highly recommended when dealing with resources.*

Syntax

```
try
{
   // risky code
}
catch(Exception e)
{
   // exception handling
}
finally
{
   // cleanup code
}
```

Example

```
class FinallyExample
{
   public static void main(String args[])
   {
      try
      {
         int a = 10;
         int b = 0;
         int c = a / b;
      }
      catch(ArithmeticException e)
      {
         System.out.println("Exception handled.");
      }
      finally
      {
         System.out.println("Program execution completed.");
      }
   }
}
```

In this program, the finally block executes whether an exception occurs or not.

Advantages of Exception Handling

1. *It prevents the program from terminating unexpectedly due to runtime errors.*
2. *It allows programmers to handle errors in a controlled and meaningful way.*
3. *It separates normal program logic from error-handling code, making programs easier to understand.*
4. *It improves program reliability and stability.*
5. *It helps programmers identify and debug runtime errors effectively.*

6. *It allows the program to continue execution even after encountering certain errors.*
7. *It supports better resource management by ensuring that important cleanup operations are executed.*

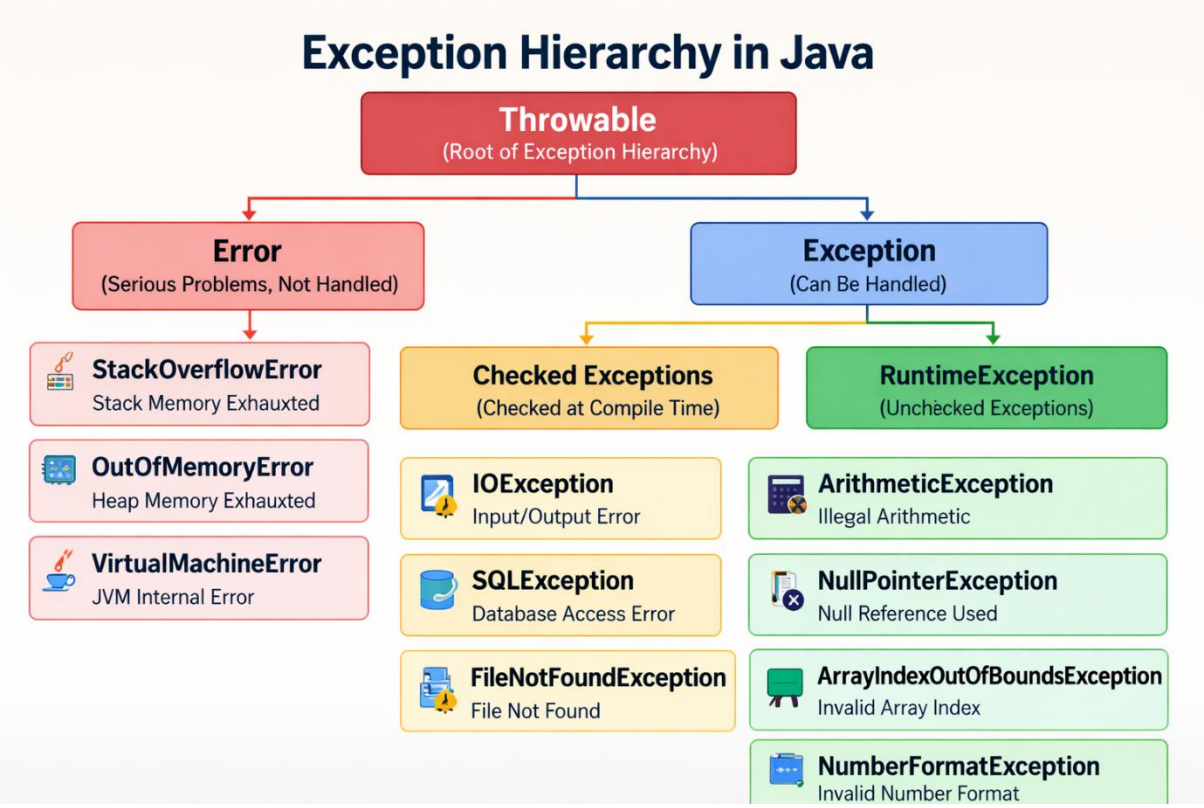
5.2 Exception Hierarchy in Java

*The **Exception Hierarchy in Java** represents the structured organization of all error and exception classes in the Java programming language. Java follows an **object-oriented programming approach**, therefore all exceptions are represented as objects and arranged in a hierarchical class structure using **inheritance**.*

This hierarchical arrangement helps the Java runtime system identify, classify, and manage different types of abnormal conditions that may occur while a program is executing. By organizing exceptions into different classes and subclasses, Java allows programmers to handle specific types of errors in a clear and systematic manner.

*In Java, every error or exception that occurs during program execution is represented by an object derived from the class **java.lang.Throwable**. This class forms the foundation of the entire exception handling mechanism.*

*The **Throwable class** is considered the **root class of the exception hierarchy**. All error and exception classes are direct or indirect subclasses of this class. Whenever a problem occurs during the execution of a Java program, an object of a subclass of Throwable is created and transferred through the program until it is handled.*



1. Throwable Class (Root of Exception Hierarchy)

*The **Throwable class** is the topmost class in Java's exception hierarchy and acts as the parent class for all types of errors and exceptions.*
*Whenever an abnormal situation arises during the execution of a program, the **Java Virtual Machine (JVM)** creates an object belonging to a subclass of the Throwable class and **throws** it. This object contains information about the error such as the type of exception, the message describing the error, and the location where the error occurred.*
The Throwable class provides several useful methods that help programmers analyze and debug problems in a program.
Some commonly used methods include:
getMessage()
This method returns a detailed message describing the exception that occurred.
toString()
This method returns the name of the exception class along with the error message.
printStackTrace()
This method prints the complete stack trace of the exception, showing the sequence of method calls that led to the error.
*The Throwable class has **two important subclasses**:*

- ***Error***
- ***Exception***

These two categories represent different kinds of abnormal conditions that may arise during program execution.

---

2. Error Class
*The **Error class** represents serious problems that occur within the **Java runtime environment** or the **Java Virtual Machine (JVM)**. These errors usually indicate critical failures that are beyond the control of application programs.*
*Errors are generally caused by problems such as insufficient memory, internal JVM failures, or system resource limitations. Because these problems are severe and usually cannot be corrected by the program itself, they are typically **not handled using try–catch blocks**.*
Most applications simply allow such errors to terminate the program, because recovery from these situations is often impossible.
Some common examples of errors include:
StackOverflowError
*This error occurs when the program uses too much stack memory. It usually happens when a method repeatedly calls itself through **infinite or deep recursion**, eventually exhausting the available stack space.*
OutOfMemoryError
This error occurs when the Java Virtual Machine cannot allocate enough memory for creating new objects. It generally happens when the program attempts to use more heap memory than is available.
VirtualMachineError
This error indicates that the JVM itself is malfunctioning or has run out of necessary resources required for operation.

*Errors represent **serious system-level problems**, and most Java programs do not attempt to handle them.*

---

3. Exception Class

*The **Exception class** represents abnormal conditions that occur during program execution but can usually be handled by the programmer. Unlike errors, exceptions are generally **recoverable**, meaning the program can take corrective action and continue its execution. Exceptions may arise due to various reasons such as invalid user input, incorrect program logic, missing files, database access failures, or network communication issues.*

*Java provides a powerful mechanism to detect these conditions and respond appropriately using **exception handling constructs** like try, catch, throw, and throws.*

*The **Exception class** is further divided into two major categories:*

- ***Checked Exceptions***
- ***Unchecked Exceptions***

5.3 Exception vs Error in Java

Basis of Comparison	***Exception***	***Error***
Definition	*An **exception** is an abnormal condition that occurs during program execution and disrupts the normal flow of the program, but it can usually be handled by the programmer.*	*An **error** is a serious problem that occurs in the system or Java Virtual Machine and is generally beyond the control of the programmer.*
Class Hierarchy	*Exceptions are subclasses of **java.lang.Exception**.*	*Errors are subclasses of **java.lang.Error**.*
Root Class	*Both exceptions and errors originate from the root class **java.lang.Throwable**.*	*Same root class **Throwable**.*
Nature of Problem	*Represents problems that occur due to logical mistakes, invalid input, or external conditions such as file handling.*	*Represents serious problems related to the system environment or JVM failures.*
Handling	*Exceptions can be handled using **try, catch, throw, and throws** mechanisms.*	*Errors are usually not handled by application programs.*
Recoverability	*Programs can often recover from exceptions and continue execution.*	*Errors are generally unrecoverable and often terminate the program.*
Occurrence	*Occurs due to issues in program logic or application-level operations.*	*Occurs due to system-level issues such as memory shortage or JVM malfunction.*
Compiler Checking	*Some exceptions (checked exceptions) are verified by the compiler.*	*Errors are not checked by the compiler.*

Examples	*ArithmeticException, IOException, NullPointerException, ArrayIndexOutOfBoundsException.*	*StackOverflowError, OutOfMemoryError, VirtualMachineError.*
Program Control	*After handling the exception, the program may continue execution normally.*	*When an error occurs, the program usually stops execution.*

5.4 Classification of Exceptions in Java

In Java programming, exceptions represent abnormal conditions that interrupt the normal flow of program execution. Since different types of problems may occur during the execution of a program, Java organizes exceptions into different categories. This systematic grouping is known as the ***classification of exceptions****.*

The classification of exceptions helps programmers understand the nature of different errors and apply suitable methods to handle them. By categorizing exceptions, Java makes it easier to identify whether an error occurs due to external factors such as file handling or network issues, or due to logical mistakes in the program itself.

All exceptions in Java are part of the exception hierarchy and are derived from the class ***java.lang.Throwable****. Under this hierarchy, the class* ***java.lang.Exception*** *represents the main category of exceptions that can be handled by programmers.*

Based on how the Java compiler treats them and how they occur during program execution, exceptions are mainly classified into two types:

1. ***Checked Exceptions***
2. ***Unchecked Exceptions***

This classification plays an important role in designing reliable and fault-tolerant Java applications.

5.4.1 Checked Exceptions

Checked exceptions are exceptions that are ***checked by the Java compiler during the compilation phase*** *of a program. Before the program is executed, the compiler verifies whether the code that may cause certain exceptions is properly handled.*

If a method contains statements that may produce a checked exception, the programmer must either:

- *handle the exception using a* ***try–catch block****, or*
- *declare the exception using the* ***throws keyword*** *in the method declaration.*

If these exceptions are not handled or declared, the compiler will produce a ***compile-time error****, preventing the program from running. Because of this strict checking mechanism, these exceptions are called* ***checked exceptions****.*

Checked exceptions generally occur when a program interacts with ***external resources*** *or performs operations that depend on factors outside the program's control. For example, reading a file from a disk, accessing a database, or communicating over a network may fail due to environmental conditions.*

Characteristics of Checked Exceptions

- *They are checked by the compiler during compilation.*
- *They must be handled using try–catch blocks or declared using the throws keyword.*
- *They usually occur due to external conditions such as file systems, input/output operations, or database connectivity.*
- *They help programmers anticipate potential problems and write safer code.*

Examples of Checked Exceptions

Some common checked exceptions include:

IOException

This exception occurs when an input or output operation fails. It may happen while reading data from a file, writing to a file, or performing network communication.

FileNotFoundException

This exception occurs when a program tries to open a file that does not exist at the specified location.

SQLException

This exception occurs when an error arises while interacting with a database using SQL statements.

ClassNotFoundException

This exception occurs when the Java Class Loader cannot locate the specified class during runtime.

Handling Checked Exceptions

Checked exceptions can be handled in two ways in Java:

1. Using Try–Catch Block

*The most common method of handling checked exceptions is by using a **try–catch block**. The risky code is placed inside the try block, and the catch block handles the exception if it occurs.*

Example:

```
import java.io.*;

class CheckedExample {
    public static void main(String[] args) {

        try {
            FileReader file = new FileReader("data.txt");
            System.out.println("File opened successfully.");
        }

        catch (FileNotFoundException e) {
            System.out.println("File not found.");
        }
    }
}
```

*In this example, the program attempts to open a file. If the file does not exist, a **FileNotFoundException** occurs and is handled by the catch block.*

2. Using the throws Keyword

Another way to handle checked exceptions is by declaring them using the ***throws keyword*** *in the method declaration. In this case, the responsibility of handling the exception is passed to the calling method.*

Example:

```
import java.io.*;

class Example {

   static void readFile() throws FileNotFoundException {
      FileReader file = new FileReader("data.txt");
   }

   public static void main(String[] args) {
      try {
         readFile();
      }

      catch (FileNotFoundException e) {
         System.out.println("Exception handled in main method.");
      }
   }
}
```

Here, the method readFile() declares the exception using the ***throws keyword****, and the exception is handled in the main method.*

5.4.2 Unchecked Exceptions

Unchecked exceptions are exceptions that are ***not checked by the compiler during compilation****. These exceptions occur during the execution of a program and are mainly caused by programming mistakes or incorrect logic in the code.*

Unchecked exceptions are subclasses of the class ***java.lang.RuntimeException****. Since these exceptions are not verified by the compiler, the programmer is not forced to handle them explicitly. However, it is considered good programming practice to manage them properly. Unchecked exceptions generally occur due to errors such as invalid arithmetic operations, incorrect array indexing, null references, or improper data type conversions.*

Characteristics of Unchecked Exceptions

- *They are not checked by the compiler at compile time.*
- *They occur during program execution.*
- *They usually arise due to logical or programming errors.*
- *Handling them is optional but recommended to improve program reliability.*

Examples of Unchecked Exceptions

Some common unchecked exceptions include:

ArithmeticException

This exception occurs when an illegal arithmetic operation is performed, such as dividing a number by zero.

NullPointerException

This exception occurs when a program attempts to access an object reference that has not been initialized.

ArrayIndexOutOfBoundsException

This exception occurs when a program tries to access an array element using an index that is outside the valid range.

NumberFormatException

This exception occurs when a string is converted into a numeric value but the string does not contain a valid number.

Example Program for Unchecked Exception

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

        int a = 10;
        int b = 0;

        int result = a / b;

        System.out.println(result);
    }
}
```

*In this program, dividing a number by zero generates an **ArithmeticException** during execution.*

Example2:

ArrayIndexOutOfBoundsException

This exception occurs when a program tries to access an array element using an index that is outside the valid range of the array.

Example:

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

        int arr[] = {10, 20, 30};

        System.out.println(arr[5]);
    }
}
```

Here, the array has only three elements, but the program attempts to access index 5, which produces an exception.

Differences Between Checked and Unchecked Exceptions

Basis of Comparison	***Checked Exceptions***	***Unchecked Exceptions***
Definition	*Checked exceptions are exceptions that are detected and verified by the Java compiler during compilation.*	*Unchecked exceptions are exceptions that are not checked by the compiler and occur during program execution.*
Compiler Checking	*The compiler checks these exceptions at compile time.*	*The compiler does not check these exceptions.*
Handling Requirement	*These exceptions must be handled using a try–catch block or declared using the throws keyword.*	*Handling these exceptions is optional for the programmer.*
Cause	*Usually occur due to external conditions such as file handling, database access, or network communication.*	*Usually occur due to logical errors or mistakes in the program code.*
Program Compilation	*If not handled properly, the program will not compile.*	*The program compiles successfully even if these exceptions are not handled.*
Class Hierarchy	*These exceptions are direct subclasses of the Exception class but not of RuntimeException.*	*These exceptions are subclasses of java.lang.RuntimeException.*
Nature	*Generally recoverable and expected during program execution.*	*Usually represent programming bugs or logical errors.*
Examples	*IOException, FileNotFoundException, SQLException, ClassNotFoundException.*	*ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, NumberFormatException.*

5.5 Re-throwing Exceptions

Introduction

Re-throwing an exception *in Java is the process of catching an exception in a catch block and then throwing the same exception again so that it can be handled by another method or higher-level exception handler. This concept is useful when a method detects an error but is not able to completely handle it by itself.*

In such cases, the method may perform some preliminary processing such as displaying an error message or logging the problem and then pass the exception to another method for further handling.

Concept of Re-throwing Exceptions

*When an exception occurs in a program, the **Java Virtual Machine (JVM)** creates an exception object derived from **java.lang.Exception**, which ultimately comes from the root class **java.lang.Throwable**.*

*Sometimes a method may catch an exception but may not be capable of resolving the problem completely. In such situations, the method can **re-throw the exception** using the throw statement. This allows the exception to move to the calling method where it can be handled properly.*

*Thus, re-throwing allows exceptions to **propagate through multiple levels of the program** until they reach an appropriate handler.*

Working Mechanism of Re-throwing Exceptions

The process of re-throwing exceptions generally follows these steps:

1. *An exception occurs inside the **try block**.*
2. *The exception is caught in a **catch block**.*
3. *Some processing such as logging or displaying a message may occur.*
4. *The exception is thrown again using the **throw statement**.*
5. *The exception is then handled by another catch block in the calling method.*

This process allows exceptions to be handled at different levels in a program.

Advantages of Re-throwing Exceptions

Re-throwing exceptions provides several benefits in Java programming.

- *It allows exceptions to be handled at different levels of a program.*
- *It supports **exception propagation** across multiple methods.*
- *It improves program design by separating **error detection** and **error handling** responsibilities.*
- *It helps maintain detailed information about where the error occurred.*
- *It improves debugging and error reporting in large applications.*

Program

```
class RethrowExample {

    static void divideNumbers() {
        try {
            int a = 20;
            int b = 0;
            int result = a / b;
            System.out.println(result);
        }
        catch (ArithmeticException e) {
            System.out.println("Exception caught in divideNumbers() method");
            throw e;   // Re-throwing the exception
        }
    }
```

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

        try {
            divideNumbers();
        }
        catch (ArithmeticException e) {
            System.out.println("Exception handled in main() method");
        }

    }
}
```

Explanation

*In this program, the method **divideNumbers()** performs a division operation. Since the value of b is zero, the statement a / b produces an **ArithmeticException**.*

1. *The exception occurs inside the **try block** of the divideNumbers() method.*
2. *The **catch block** catches the exception and prints a message.*
3. *Inside the catch block, the exception is **re-thrown** using the throw statement.*
4. *The exception is then passed to the **main() method**.*
5. *The catch block in the **main() method** finally handles the exception and displays another message.*

*Thus, the same exception is **caught in one method and thrown again to be handled by another method**, which demonstrates the concept of **re-throwing exceptions**.*

5.6 Types of Exceptions in Java

*In Java programming, an **exception** is an event that occurs during the execution of a program and interrupts the normal sequence of instructions. Such events represent abnormal conditions that prevent a program from performing its intended task. These conditions may arise due to various reasons such as invalid user input, arithmetic errors, file handling problems, memory limitations, or logical mistakes in the program.*

*To manage such situations effectively, Java provides a structured **exception handling mechanism**. As part of this mechanism, Java organizes exceptions into different categories so that programmers can understand the source of the problem and handle it appropriately. This systematic grouping of exceptions is known as the **types of exceptions in Java**.*

*All exceptions in Java belong to a hierarchical class structure derived from the root class **java.lang.Throwable**. This class represents any condition that can occur during program execution and that can be reported or handled by the Java runtime system.*

*Under this hierarchy, the class **java.lang.Exception** represents exceptions that can be handled by application programs. Based on their origin and purpose, exceptions in Java are generally divided into two main types:*

1. ***Built-in Exceptions***

2. ***Custom Exceptions***

5.6.1 Built-in Exceptions

Concept of Built-in Exceptions

Built-in exceptions *are the exceptions that are already defined and provided by the Java programming language. These exceptions are part of the Java standard library and are automatically generated by the* ***Java Virtual Machine (JVM)*** *when certain abnormal conditions occur during program execution.*

In real-world programming, many types of errors occur repeatedly in different programs. For example, dividing a number by zero, accessing an array element outside its valid range, attempting to open a file that does not exist, or converting an invalid string into a number. To handle such commonly occurring problems, Java provides predefined exception classes known as built-in exceptions.

Because these exceptions are already implemented in Java, programmers do not need to create new classes to represent these common error conditions. Instead, they can directly use the built-in exception classes to detect and manage runtime problems.

Built-in exceptions belong to the exception hierarchy derived from ***java.lang.Exception****, which itself extends the root class* ***java.lang.Throwable****.*

Role of Built-in Exceptions in Java Programs

Built-in exceptions play an important role in improving the reliability and stability of Java programs. When a program performs operations that may lead to abnormal situations, the Java runtime system automatically detects the problem and creates an object of the corresponding exception class.

This exception object contains detailed information about the error, such as the type of exception, the message describing the error, and the location where the problem occurred. Once the exception is generated, it can be handled using the exception handling constructs provided by Java, such as ***try, catch, throw, throws, and finally****.*

Through this mechanism, built-in exceptions allow programs to continue execution in a controlled manner instead of terminating abruptly.

Characteristics of Built-in Exceptions

Built-in exceptions possess several characteristics that make them essential for effective error handling in Java.

- *They are* ***predefined classes included in the Java standard library****.*
- *They are automatically generated by the* ***Java runtime system*** *when abnormal conditions occur.*
- *They represent* ***common programming and runtime errors****.*
- *They follow the* ***exception hierarchy derived from the Throwable class****.*
- *They can be handled using* ***try–catch blocks****.*
- *They help improve* ***program reliability, stability, and maintainability****.*

Categories of Built-in Exceptions

Built-in exceptions are further divided into two categories based on how the Java compiler treats them.

Checked Built-in Exceptions

*Checked built-in exceptions are exceptions that are **checked by the compiler during the compilation process**. The compiler ensures that these exceptions are properly handled before the program is executed.*

*If a method contains code that may generate a checked exception, the programmer must either handle the exception using a **try–catch block** or declare the exception using the **throws keyword**. If these exceptions are not handled, the program will not compile.*

*Checked exceptions usually occur during operations involving **external resources**, such as file handling, database access, and network communication.*

Examples include:

- *IOException*
- *FileNotFoundException*
- *SQLException*
- *ClassNotFoundException*

These exceptions represent situations where the program's execution may fail due to external conditions.

Unchecked Built-in Exceptions

*Unchecked built-in exceptions are exceptions that are **not checked by the compiler during compilation**. These exceptions occur during program execution and usually result from logical errors or incorrect programming practices.*

*Unchecked exceptions are subclasses of **java.lang.RuntimeException**.*

Because these exceptions are not verified by the compiler, programmers are not required to handle them explicitly. However, handling them properly helps improve the robustness of the program.

Examples include:

- *ArithmeticException*
- *NullPointerException*
- *ArrayIndexOutOfBoundsException*
- *NumberFormatException*

These exceptions generally indicate mistakes in program logic that should be corrected by the programmer.

5.6.2 Custom Exceptions

Introduction

*In Java programming, there may be situations where the predefined exceptions provided by Java are not sufficient to represent certain application-specific errors. In such cases, programmers can create their own exceptions. These programmer-defined exceptions are known as **custom exceptions** or **user-defined exceptions**.*

*A **custom exception** is an exception class created by the programmer to represent specific error conditions that are unique to a particular application. By defining custom exceptions, programmers can provide meaningful descriptions of errors that occur in their programs.*
*Custom exceptions are typically created by extending the class **java.lang.Exception**, which itself is derived from the root class **java.lang.Throwable**. In some cases, custom exceptions may also extend **java.lang.RuntimeException** if they represent unchecked exceptions.*

Concept of Custom Exceptions

*The main purpose of custom exceptions is to represent **application-specific problems** that cannot be clearly expressed using Java's built-in exceptions.*
For example, in a banking application, a program may need to generate exceptions such as:

- *InsufficientBalanceException*
- *InvalidAccountException*
- *TransactionLimitExceededException*

*Similarly, in a student management system, exceptions such as **InvalidMarksException** or **AttendanceShortageException** may be required.*
Since these types of errors are specific to particular applications, programmers create custom exception classes to represent them.

Characteristics of Custom Exceptions

Custom exceptions have several important characteristics:

- *They are **created by programmers** to represent specific application errors.*
- *They are defined as **separate classes**.*
- *They extend the **Exception class or RuntimeException class**.*
- *They provide **meaningful error messages** related to the application.*
- *They improve **program readability and maintainability**.*

Steps to Create a Custom Exception

The process of creating a custom exception generally involves the following steps:

1. *Create a new class that extends the **Exception class**.*
2. *Define a constructor in the class to pass an error message.*
3. *Use the **throw keyword** to generate the custom exception when the required condition occurs.*
4. *Handle the exception using a **try–catch block**.*

Example of Custom Exception

```
class InvalidAgeException extends Exception
{
   InvalidAgeException(String message)
   {
      super(message);
   }
}
```

```
class CustomExceptionExample
{
   static void checkAge(int age) throws InvalidAgeException
   {
      if(age < 18)
      {
         throw new InvalidAgeException("Age must be 18 or above.");
      }
      else
      {
         System.out.println("Eligible for voting.");
      }
   }

   public static void main(String args[])
   {
      try
      {
         checkAge(16);
      }
      catch(InvalidAgeException e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```

Explanation

*In this program, a custom exception class called **InvalidAgeException** is created by extending the Exception class. The method checkAge() checks whether the age is less than 18. If the age is below 18, the program throws the custom exception using the **throw** statement. The exception is then caught and handled in the main() method using a **catch block**, which displays the error message.*

Advantages of Custom Exceptions

Custom exceptions provide several advantages in Java programming:

- *They allow programmers to define **application-specific error conditions**.*
- *They improve **code readability** by providing meaningful exception names.*
- *They help in **better debugging and error reporting**.*
- *They make large applications **more structured and maintainable**.*

Smart Study
PLUS

UNIT – III

4.

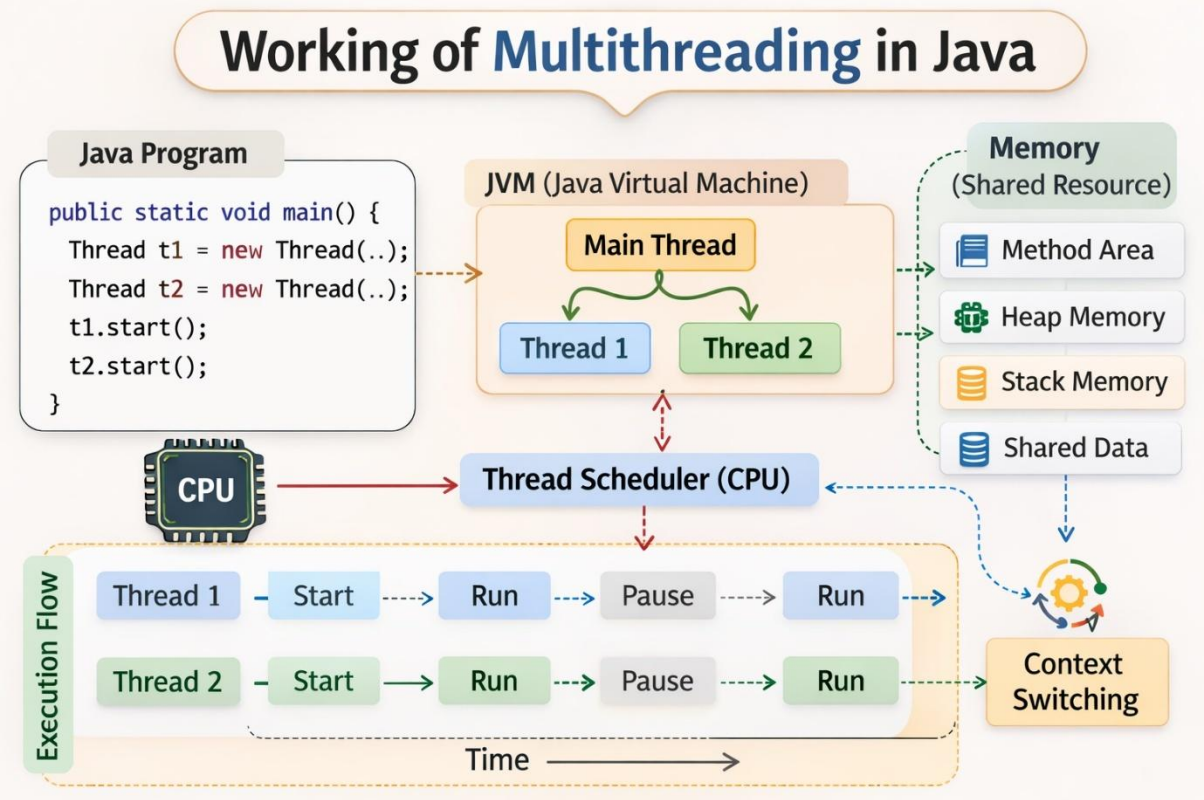
1. Multithreading in Java

Introduction

In modern computing environments, applications are expected to perform multiple operations at the same time while maintaining high performance and responsiveness. For example, a web browser may download files, display web pages, process user input, and play multimedia simultaneously. Performing these operations sequentially in a single execution path would significantly reduce system efficiency. To address this challenge, programming languages provide mechanisms that allow multiple tasks to execute concurrently.

Multithreading *is one such mechanism in Java that enables a program to perform several tasks simultaneously. It refers to the process of executing multiple threads concurrently within a single program. A* ***thread*** *is the smallest unit of execution within a process, representing an independent path of control in a program.*

In a single-threaded program, all instructions are executed sequentially. This means that if one operation takes a long time to complete, the entire program must wait until that operation finishes. Multithreading eliminates this limitation by allowing different tasks to run independently and concurrently.



Java provides built-in support for multithreading through the ***java.lang*** *package. The most important components used for creating and managing threads are* ***java.lang.Thread*** *and* ***java.lang.Runnable****.*

*Multithreading improves **CPU utilization, program efficiency, and responsiveness**, making it an essential concept in modern Java applications such as web servers, real-time systems, gaming applications, and distributed systems.*

1.1 Java Thread Model

*The **Java Thread Model** defines the framework through which threads are created, managed, scheduled, and executed in a Java program. It describes how multiple threads operate within the same program and how they interact with system resources.*

*Java follows a **multithreaded execution model**, which means that several threads can execute concurrently within a single process. Each thread represents a separate flow of execution but shares the same memory space and system resources of the process.*

This shared environment enables efficient communication between threads but also introduces challenges such as synchronization and resource management. Therefore, the Java thread model provides mechanisms to ensure safe and coordinated access to shared data.

*One of the most important aspects of the Java thread model is **thread scheduling**. Since multiple threads may be ready to execute at the same time, the **Java Virtual Machine (JVM)** determines which thread should run at a given moment. The JVM allocates CPU time among competing threads based on scheduling algorithms and thread priorities.*

*The Java thread model also supports the concept of **lightweight processes**. Threads require significantly fewer resources compared to full processes because they share memory and system resources with other threads within the same program.*

Important responsibilities of the Java thread model include:

- *Creating and initializing threads*
- *Managing thread execution*
- *Allocating CPU time among threads*
- *Synchronizing access to shared resources*
- *Facilitating communication between threads*

These mechanisms allow Java programs to execute multiple tasks efficiently while maintaining system stability.

1.2 Thread

*A **thread** is the smallest sequence of programmed instructions that can be executed independently by the Java Virtual Machine. It represents a single path of execution within a program.*

In a multithreaded program, several threads may run simultaneously, each performing a specific task. Although threads execute independently, they share the same memory space and resources of the process to which they belong.

*Every Java program automatically starts with at least one thread known as the **main thread**, which begins execution from the main() method. Additional threads can be created to perform other tasks concurrently.*

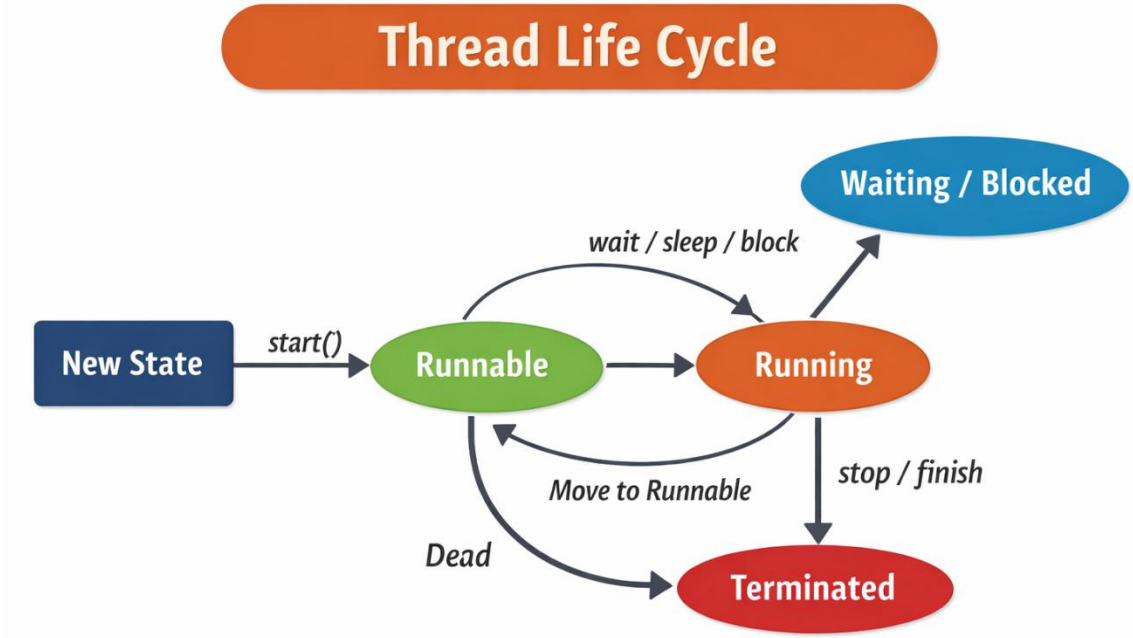
*Threads are particularly useful in applications that require **parallel or concurrent processing**. Some common real-world examples include:*

- *A web server handling multiple client requests simultaneously.*
- *A word processor performing spell checking while the user types.*
- *A multimedia application playing audio and video concurrently.*
- *An online game processing multiple player actions at the same time.*

Because threads share memory and resources, they can communicate easily with one another. However, this shared environment can also lead to problems such as ***race conditions****, where multiple threads attempt to modify the same data simultaneously.*
To prevent such issues, Java provides synchronization mechanisms that allow threads to coordinate their access to shared resources.

1.3 Thread Life Cycle

The ***thread life cycle*** *describes the sequence of states through which a thread passes from its creation until it completes execution. Each state represents a particular stage in the execution of a thread.*
Understanding the thread life cycle helps programmers control thread behavior and design efficient multithreaded applications.



New State

When a thread object is created using the Thread class, it enters the ***new state****. In this stage, the thread has been initialized but has not yet started executing.*

Runnable State

After the start() method is called, the thread enters the ***runnable state****. In this state, the thread is ready to run but is waiting for CPU time from the thread scheduler.*
Multiple threads may remain in the runnable state simultaneously while waiting for execution.

Running State

*When the thread scheduler selects a thread from the runnable pool and assigns CPU time to it, the thread enters the **running state**. During this stage, the thread executes the instructions defined in its run() method.*

Waiting or Blocked State

*A thread may temporarily stop execution and enter the **waiting or blocked state** under certain conditions. This can occur when the thread waits for another thread to release a resource, waits for input/output operations to complete, or is placed in sleep mode using the sleep() method.*

Terminated State

*Once the thread completes the execution of its run() method or is stopped due to an error, it enters the **terminated state**. At this stage, the thread has finished its execution and cannot be restarted.*

1.4 Main Thread

Introduction

*In every Java program, execution begins from a special thread called the **main thread**. When a Java application starts, the **Java Virtual Machine (JVM)** automatically creates this thread and uses it to execute the main() method. Since the main() method is the starting point of a Java program, the main thread becomes the **first thread of execution** in the application.*

The main thread plays a central role in Java programs because it controls the overall execution of the program and can create additional threads for performing different tasks concurrently.

Concept of the Main Thread

*The **main thread** is the primary thread that begins the execution of a Java program. It is automatically created by the JVM when the program starts and is responsible for executing the statements inside the main() method.*

When a Java program is launched, the JVM performs the following steps:

1. *It loads the class that contains the main() method.*
2. *It creates the **main thread**.*
3. *The main thread begins executing the statements inside the main() method.*

*Therefore, the main thread acts as the **starting point of program execution**.*

*Once the main thread begins execution, it can create other threads to perform separate tasks. These threads are often referred to as **child threads** because they are created from the main thread.*

Role of the Main Thread in Multithreading

In a multithreaded Java program, the main thread performs an important role in managing other threads. It can create multiple threads that execute different parts of the program simultaneously.

For example, a program may require different tasks such as:

- *reading input from the user*
- *performing calculations*

- *saving data to a file*
- *updating the user interface*

Instead of performing all these tasks sequentially, the main thread can create separate threads to handle each task concurrently.

*Thus, the main thread acts as the **controller that initiates and coordinates multithreaded execution**.*

Characteristics of the Main Thread

The main thread has several important characteristics:

- *It is **automatically created by the JVM** when the program starts.*
- *It executes the **main() method** of the program.*
- *It is the **first thread** that begins execution in a Java application.*
- *It can create **additional threads** for performing concurrent tasks.*
- *It controls the **overall flow of execution** of the program.*

Because of these characteristics, the main thread serves as the foundation of multithreaded programming in Java.

Relationship Between Main Thread and Other Threads

*In a multithreaded program, the main thread often acts as the **parent thread**, while the threads it creates are called **child threads**.*

The main thread may perform tasks such as:

- *creating new thread objects*
- *starting thread execution*
- *controlling thread priorities*
- *coordinating thread completion*

Once child threads are created, they execute independently, but they still share the same memory space and system resources with the main thread.

This shared environment allows threads to communicate efficiently, but it also requires proper synchronization to prevent conflicts.

Accessing the Main Thread

*Java allows programmers to obtain information about the currently executing thread using the currentThread() method of the **Thread class** available in the **java.lang**.*

Example:

```
class MainThreadExample
{
   public static void main(String args[])
   {
      Thread t = Thread.currentThread();
      System.out.println("Current Thread: " + t);
   }
}
```

This program retrieves the reference of the currently executing thread, which is the main thread, and displays its information.

1.5 Creating Threads in Java

Introduction

Multithreading is one of the most powerful features of Java that allows a program to execute multiple tasks concurrently. In order to perform multithreading, it is necessary to create ***threads*** *that represent independent paths of execution within a program.*

A thread performs a specific task while sharing the resources of the program with other threads. For example, a web browser may create different threads for downloading files, displaying images, and processing user input. Each of these activities is handled by a separate thread, allowing them to run simultaneously.

Java provides strong built-in support for thread creation and management through the ***java.lang*** *package. The two most commonly used components for creating threads are the* ***java.lang.Thread*** *class and the* ***java.lang.Runnable*** *interface.*

In Java, threads can be created using two primary approaches:

1. ***Implementing the Runnable Interface***
2. ***Extending the Thread Class***

Both approaches ultimately involve defining the code that will be executed by the thread inside a method called run(). When the thread is started using the start() method, the Java Virtual Machine creates a new execution path and invokes the run() method automatically.

1.5.1 Implementing Runnable Interface

Concept

One of the most commonly used and recommended methods for creating threads in Java is by implementing the ***Runnable interface****. The Runnable interface represents a task that can be executed by a thread.*

The Runnable interface contains only a single method called run(). This method defines the behavior of the thread and contains the code that will be executed when the thread runs.

When a class implements the Runnable interface, it provides the implementation of the run() method. An object of this class is then passed to a Thread object. When the thread's start() method is called, the JVM begins executing the run() method of the Runnable object in a separate thread.

Conceptual Working of Runnable-Based Threads

The process of creating and executing a thread using the Runnable interface can be understood conceptually in the following way:

1. *A class defines the task by implementing the Runnable interface.*
2. *The run() method contains the instructions that the thread will execute.*
3. *An object of the class is created.*
4. *A Thread object is created and associated with the Runnable object.*
5. *When the start() method is invoked, the JVM creates a new thread and executes the run() method.*

*This approach separates the **task definition** from the **thread execution mechanism**, making the program more flexible and easier to manage.*

Advantages of Using Runnable Interface

The Runnable approach offers several advantages:

- *It allows a class to extend another class while still implementing Runnable, since Java supports multiple interfaces but not multiple inheritance of classes.*
- *It separates the **task to be executed** from the **thread that executes the task**.*
- *It promotes better object-oriented design and code reusability.*
- *It is generally considered the **preferred method for creating threads in Java**.*

Example

```
class MyThread implements Runnable
{
   public void run()
   {
      System.out.println("Thread created using Runnable interface");
   }

   public static void main(String args[])
   {
      MyThread obj = new MyThread();
      Thread t = new Thread(obj);
      t.start();
   }
}
```

In this example, the class implements the Runnable interface and defines the code that the thread will execute inside the run() method.

1.5.2 Extending Thread Class

Concept

*Another way to create a thread in Java is by extending the **Thread class**. In this method, a new class is derived from the Thread class and the run() method is overridden to define the thread's behavior.*

The Thread class represents a thread of execution in Java. By extending this class, a programmer can create a new thread type that contains its own execution logic.

When an object of the subclass is created and the start() method is called, the JVM automatically creates a new thread and invokes the overridden run() method.

Conceptual Working of Thread-Class-Based Threads

The process of creating a thread using the Thread class involves the following steps:

1. *A new class is created by extending the Thread class.*
2. *The run() method is overridden to specify the thread's task.*

3. *An object of the class is created.*
4. *The start() method is called to begin execution.*
5. *The JVM creates a new thread and invokes the run() method.*

*This approach combines both the **thread definition** and **thread execution logic** in the same class.*

Limitations of Extending Thread Class

Although this approach is simple and easy to understand, it has some limitations:

- *Java does not support **multiple inheritance of classes**. If a class extends the Thread class, it cannot extend any other class.*
- *It tightly couples the task with the thread implementation, reducing flexibility.*

Because of these limitations, this method is often used in simpler programs, while the Runnable approach is preferred in larger applications.

Example

```
class MyThread extends Thread
{
   public void run()
   {
      System.out.println("Thread created by extending Thread class");
   }

   public static void main(String args[])
   {
      MyThread t = new MyThread();
      t.start();
   }
}
```

In this example, the class extends the Thread class and overrides the run() method to define the thread's behavior.

1.6 Multiple Threads

Introduction

*In Java, **multiple threads** refer to the execution of two or more threads concurrently within a single program. Multithreading allows a program to perform several tasks simultaneously, thereby improving the efficiency and responsiveness of the application. Each thread represents an independent path of execution, but all threads operate within the same process and share the same memory space and system resources.*

The concept of multiple threads is fundamental to modern software systems because many real-world applications must perform several operations at the same time. For instance, a web browser may download files, display images, play audio, and respond to user input simultaneously. Such functionality is made possible through the use of multiple threads.

*Java provides extensive support for multithreading through classes and interfaces available in the **java.lang** package. The **java.lang.Thread** class and the **java.lang.Runnable** interface are commonly used to create and manage threads.*

Concept of Multiple Threads

The concept of multiple threads is based on the idea that a single program can contain several independent units of execution that run concurrently. Each thread performs a specific task while sharing the resources of the program with other threads.

*In a multithreaded environment, threads execute concurrently rather than sequentially. Although it may appear that threads are running simultaneously, the **Java Virtual Machine (JVM)** actually schedules and allocates CPU time among threads very rapidly. This scheduling creates the illusion that multiple threads are executing at the same time.*

Multiple threads within the same program share several important resources, including:

- *Memory space*
- *Program code*
- *Open files*
- *Input and output devices*

Because threads share these resources, communication between them becomes efficient. However, this shared environment also requires proper coordination to prevent conflicts when multiple threads attempt to access the same resource simultaneously.

Working of Multiple Threads

*The execution of multiple threads in Java is controlled by the **thread scheduler** of the Java Virtual Machine. The thread scheduler determines which thread should run at a given moment.*

*When several threads are ready for execution, they enter the **runnable state**. The JVM selects one of these threads and assigns CPU time to it. After a short period of execution, the CPU may switch to another thread. This process is known as **context switching**.*

Through rapid context switching, the JVM ensures that all threads receive an opportunity to execute, giving the appearance of simultaneous execution.

Characteristics of Multiple Threads

Multiple threads in Java have several important characteristics:

- *They allow **concurrent execution of tasks** within a single program.*
- *All threads share the **same memory and system resources**.*
- *Each thread has its own **execution path and stack**.*
- *Threads can communicate and coordinate with each other.*
- *The **JVM manages thread scheduling and execution**.*

These characteristics enable Java programs to perform complex operations efficiently.

Advantages of Multiple Threads

Improved Performance

Multiple threads allow a program to divide complex tasks into smaller units that can be executed concurrently. This improves the overall performance of the application.

Better CPU Utilization

Threads help utilize CPU resources efficiently. When one thread is waiting for input/output operations, another thread can continue executing.

Enhanced Responsiveness

Applications remain responsive even when performing time-consuming operations. For example, a graphical application can respond to user input while performing background processing.

Efficient Resource Sharing

Since threads share memory and system resources, communication between them is faster and more efficient compared to communication between separate processes.

Simplified Program Structure

Complex tasks can be divided into smaller threads, making programs easier to design and maintain.

Example of Multiple Threads in Java

```
class MyThread extends Thread
{
   public void run()
   {
      for(int i = 1; i <= 5; i++)
      {
         System.out.println(Thread.currentThread().getName() + " : " + i);
      }
   }

   public static void main(String args[])
   {
      MyThread t1 = new MyThread();
      MyThread t2 = new MyThread();
      MyThread t3 = new MyThread();

      t1.start();
      t2.start();
      t3.start();
   }
}
```

Explanation

In this example, three thread objects (t1, t2, and t3) are created. When the start() method is invoked for each thread, the JVM creates three separate threads that execute the run() method concurrently.

Each thread prints numbers independently, demonstrating the concept of multiple threads executing simultaneously.

1.7 Thread Priorities

Introduction

In a multithreaded Java program, multiple threads may be ready to execute at the same time. Since a CPU can execute only one thread at a given moment, the system must decide ***which thread should run first****. This decision is influenced by several factors, one of the most important being* ***thread priority****.*

Thread priority is a mechanism that indicates the ***relative importance of a thread*** *compared to other threads in a program. It helps the system determine the order in which threads should be executed when several threads are competing for CPU time.*

Java provides support for assigning and managing thread priorities through the ***java.lang.Thread*** *class, which is available in the* ***java.lang****.*

Concept of Thread Priority

Thread priority represents the ***importance level assigned to a thread****. When several threads are in the runnable state and waiting for CPU time, the* ***thread scheduler*** *of the Java Virtual Machine may give preference to the thread with a higher priority.*

The thread scheduler is responsible for managing the execution of threads. It selects one thread from the pool of runnable threads and allocates CPU time to it. Thread priority helps the scheduler decide which thread should be executed first.

However, thread priority does not guarantee that a higher-priority thread will always execute before a lower-priority thread. The final scheduling decision depends on the ***operating system and JVM implementation****.*

Thus, thread priority acts as a ***suggestion to the scheduler rather than a strict rule****.*

Priority Levels in Java

In Java, each thread is assigned a priority value that ranges from ***1 to 10****.*

- ***Priority 1*** *represents the lowest priority.*
- ***Priority 10*** *represents the highest priority.*

Threads with higher priority are generally given more opportunities to execute than threads with lower priority.

Java defines three commonly used priority constants:

- ***MIN_PRIORITY (1)*** *– Represents the lowest priority.*
- ***NORM_PRIORITY (5)*** *– Represents the default priority assigned to threads.*
- ***MAX_PRIORITY (10)*** *– Represents the highest priority.*

When a new thread is created, it normally inherits the priority of the thread that created it. In most cases, this means that newly created threads start with the default priority level.

Role of Thread Priority in Scheduling

Thread priority plays an important role in ***thread scheduling****, which is the process of allocating CPU time among multiple threads.*

When several threads are ready to execute, the scheduler may perform the following actions:

1. *Identify all threads in the **runnable state**.*
2. *Compare their priority values.*
3. *Select a thread with higher priority for execution.*
4. *Allocate CPU time to the selected thread.*

*If multiple threads have the same priority level, the scheduler may choose one of them using techniques such as **round-robin scheduling** or time sharing.*

Importance of Thread Priority

Thread priority is useful in situations where certain tasks are more important than others. By assigning higher priority to critical threads, programmers can ensure that important operations receive more CPU attention.

For example:

- *A thread responsible for **user interaction** may have higher priority.*
- *A background thread performing **data logging** may have lower priority.*

By carefully managing thread priorities, programmers can design applications that are more efficient and responsive.

1.8 Synchronization

Introduction

In multithreaded programming, several threads may execute simultaneously within the same program. Since these threads operate within the same process, they often share common resources such as objects, variables, files, and memory areas. While this shared environment improves efficiency and communication between threads, it can also lead to serious problems when multiple threads attempt to access or modify the same resource at the same time.

*To prevent such conflicts and ensure the correct execution of programs, Java provides a mechanism known as **synchronization**. Synchronization is a technique used to control the access of multiple threads to shared resources so that only one thread can execute a critical section of code at a time.*

*Java supports synchronization using features provided by classes in the **java.lang**, particularly through the **java.lang.Thread** class and the synchronized keyword.*

*Synchronization plays a vital role in ensuring **data consistency, thread coordination, and reliable execution** of multithreaded programs.*

Synchronization

*Synchronization refers to the process of **controlling the execution of multiple threads so that shared resources are accessed in a safe and orderly manner**.*

When multiple threads run concurrently, they may attempt to read or modify the same data simultaneously. If proper coordination is not maintained, the final result may become unpredictable or incorrect.

For example, consider two threads that attempt to update the value of a shared variable at the same time. If both threads read the variable simultaneously and modify it independently, the final value may not reflect the intended result. This situation occurs because the operations of the threads overlap with each other.

*Synchronization prevents this problem by ensuring that **only one thread can access the shared resource at any given time**, while other threads must wait until the resource becomes available.*

Race Condition

*One of the major problems that occurs in multithreaded programs without synchronization is known as a **race condition**.*

A race condition occurs when two or more threads attempt to access and modify the same shared data simultaneously, and the final outcome depends on the unpredictable timing of thread execution.

Since thread execution order cannot always be controlled, race conditions may lead to inconsistent or incorrect results.

Synchronization eliminates race conditions by allowing only one thread to execute the critical section at a time.

Critical Section

*The portion of a program where shared resources are accessed or modified is known as the **critical section**.*

Because multiple threads may attempt to execute the critical section simultaneously, it must be protected using synchronization. When a thread enters the critical section, other threads must wait until the current thread finishes executing that section.

By protecting critical sections, synchronization ensures that shared data remains consistent and reliable.

How Synchronization Works

*In Java, synchronization is commonly implemented using the **synchronized keyword**. When a method or block of code is declared as synchronized, it ensures that only one thread can execute that code at a time.*

*Internally, the Java Virtual Machine uses a mechanism known as a **monitor lock** to implement synchronization.*

The working process can be understood as follows:

1. *Each object in Java is associated with a monitor lock.*
2. *When a thread enters a synchronized method or block, it acquires the lock for that object.*
3. *While the lock is held by one thread, other threads attempting to access the same synchronized code must wait.*
4. *When the thread finishes executing the synchronized code, it releases the lock.*
5. *Another waiting thread can then acquire the lock and execute the code.*

This locking mechanism ensures that shared resources are accessed safely without interference from other threads.

Types of Synchronization

Synchronization in Java can be achieved in different ways.

Synchronized Method

A method can be declared synchronized so that only one thread can execute it at a time for a given object.

Example:

```
synchronized void display()
{
   // critical section
}
```

When a thread calls this method, it must first acquire the object's monitor lock.

Synchronized Block

Instead of synchronizing an entire method, it is also possible to synchronize only a specific portion of code.

Example:

```
synchronized(object)
{
   // critical section
}
```

This approach improves efficiency because only the necessary part of the code is protected.

Advantages of Synchronization

Synchronization provides several important benefits in multithreaded programming:

Data Consistency

It ensures that shared data remains accurate and consistent even when multiple threads access it.

Prevention of Race Conditions

Synchronization eliminates race conditions by allowing only one thread to execute the critical section at a time.

Thread Coordination

It helps coordinate the execution of multiple threads, ensuring orderly access to shared resources.

Reliable Program Execution

By preventing conflicts between threads, synchronization improves the reliability and correctness of multithreaded programs.

Limitations of Synchronization

Although synchronization is necessary for safe multithreaded execution, excessive use of synchronization may also create certain challenges.

- *It may reduce program performance because threads must wait for locks to be released.*
- *It may cause **deadlock**, where two or more threads wait indefinitely for each other to release resources.*
- *It increases the complexity of program design.*

Therefore, synchronization should be used carefully and only when necessary.

Conceptual Understanding

*From a conceptual perspective, synchronization acts as a **traffic control system for threads**. Just as traffic signals control the movement of vehicles to prevent collisions, synchronization controls the execution of threads to prevent conflicts when accessing shared resources.*
By ensuring that threads execute critical sections in an orderly manner, synchronization maintains the integrity and reliability of multithreaded applications.
Thus, synchronization is a fundamental concept in Java multithreading that enables safe and coordinated execution of multiple threads within a program.

Example of Synchronization

```
class Counter
{
   int count = 0;

   synchronized void increment()
   {
      count++;
   }
}

class SyncExample
{
   public static void main(String args[])
   {
      Counter c = new Counter();

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

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

      t1.start();
      t2.start();
   }
}
```

Explanation

In this program, two threads attempt to increment the same variable count. If the method were not synchronized, both threads might modify the variable at the same time, producing incorrect results.
*By declaring the increment() method as **synchronized**, only one thread can execute it at a time, ensuring that the value of count is updated correctly.*

2. Collections Framework
Introduction
*The **Collections Framework** in Java is a unified architecture that provides a set of classes and interfaces for storing, managing, and manipulating groups of objects. In many real-world applications, programs need to handle collections of data such as lists of students, sets of numbers, or mappings between keys and values. The Collections Framework provides efficient and flexible ways to perform these operations.*
Before the introduction of the Collections Framework, Java used older classes such as Vector, Stack, Hashtable, and Array. These classes were not part of a unified structure and had limitations in terms of flexibility and efficiency. To overcome these limitations, the Collections Framework was introduced to provide a standardized approach to managing groups of objects.
*The Collections Framework is part of the **java.util** package and includes several important interfaces and classes such as **java.util.List**, **java.util.Set**, **java.util.Map**, and **java.util.Collection**.*
The main purpose of the Collections Framework is to simplify the development process by providing ready-made data structures and algorithms for handling collections of objects.

Concept of Collections Framework
*The Collections Framework can be understood as a **framework that organizes different data structures and algorithms into a well-defined hierarchy of interfaces and classes**. It allows programmers to store and manipulate objects efficiently without implementing complex data structures from scratch.*
In this framework, objects are stored as elements within collection classes. These classes provide methods for performing various operations such as:

- *Adding elements*
- *Removing elements*
- *Searching elements*
- *Sorting elements*
- *Iterating through elements*

The framework also provides algorithms for common operations such as sorting, searching, and data manipulation.

Components of the Collections Framework
The Java Collections Framework mainly consists of three important components:

- ***Interfaces***
- ***Classes (Implementation Classes)***

- ***Algorithms***
- ***Iterators***

2.2 Collection Interface

Introduction

*The **Collection Interface** is the root interface of the Java Collections Framework. It represents a group of objects, known as elements, that can be stored and manipulated as a single unit. The Collection interface defines the fundamental operations that can be performed on collections, such as adding elements, removing elements, and searching for elements.*

*The Collection interface is part of the **java.util** and serves as the base interface for many other collection interfaces such as **java.util.List**, **java.util.Set**, and **java.util.Queue**. These interfaces extend the **java.util.Collection** and provide specialized ways of storing and managing objects.*

*The main purpose of the Collection interface is to provide a **common framework for representing and manipulating groups of objects** in a consistent manner.*

Concept of Collection Interface

*Conceptually, the Collection interface can be understood as a **container that holds multiple objects**. Instead of handling objects individually, a collection allows programmers to group objects together and perform operations on them collectively.*

The Collection interface itself does not define how elements are stored internally. Instead, it defines a set of methods that allow programmers to manipulate collections. Different classes implement the Collection interface using different internal data structures.

For example:

- *A **List** maintains elements in an ordered sequence.*
- *A **Set** stores unique elements and does not allow duplicates.*
- *A **Queue** manages elements in a specific order for processing.*

Although these structures behave differently, they all follow the operations defined by the Collection interface.

Characteristics of the Collection Interface

The Collection interface has several important characteristics.

*First, it allows the storage of **multiple objects within a single container**, making it easier to manage groups of data.*

*Second, collections are **dynamic in nature**. Unlike arrays, collections can grow or shrink automatically as elements are added or removed.*

*Third, the Collection interface provides **standard methods for performing common operations** such as adding, removing, and searching elements.*

*Another important characteristic is that collections support **iteration**, which allows elements to be accessed sequentially using iterators.*

*Finally, the Collection interface forms the **foundation of the collections hierarchy**, ensuring that all collection classes follow a common structure.*

Important Methods of the Collection Interface

The Collection interface defines several methods that are commonly used to manipulate collections.

- ***add()*** *– Adds an element to the collection.*
- ***remove()*** *– Removes an element from the collection.*
- ***size()*** *– Returns the number of elements in the collection.*
- ***isEmpty()*** *– Checks whether the collection is empty.*
- ***contains()*** *– Checks whether a specified element exists in the collection.*
- ***clear()*** *– Removes all elements from the collection.*

These methods allow programmers to manage collections efficiently.

Example of Collection Interface

Example program using a collection class that implements the Collection interface:

```
import java.util.ArrayList;
import java.util.Collection;

public class CollectionExample
{
   public static void main(String[] args)
   {
      Collection<String> students = new ArrayList<>();

      students.add("Ravi");
      students.add("Anita");
      students.add("Rahul");

      System.out.println("Students List: " + students);
      System.out.println("Total Students: " + students.size());

      students.remove("Rahul");

      System.out.println("After Removing: " + students);
      System.out.println("Contains Ravi? " + students.contains("Ravi"));
   }
}
```

Explanation

In this example, an ***ArrayList*** *object is created using the* ***Collection interface reference****. Elements representing student names are added to the collection using the add() method. The size() method is used to determine the number of elements in the collection. The remove() method deletes an element, and the contains() method checks whether a specific element exists in the collection.*

This example demonstrates how the Collection interface provides standard methods for managing groups of objects.

2.3 List Interface

Introduction

*The **List Interface** is an important interface in the Java Collections Framework that represents an **ordered collection of elements**. In a list, elements are stored in a sequential order, and each element can be accessed by its position in the collection. This position is known as the **index**.*

*The List interface extends the **java.util.Collection** and is available in the **java.util**. It defines several methods that allow programmers to insert, remove, update, and retrieve elements from a list.*

*Several classes implement the List interface, including **java.util.ArrayList**, **java.util.LinkedList**, and Vector. These classes provide different internal implementations for storing elements while following the rules defined by the List interface.*

Concept of List Interface

*The List interface represents a **collection in which elements are arranged in a specific order**. Unlike some other collection types, a list maintains the order in which elements are inserted. This means that when elements are added to the list, they are stored in the same sequence.*

*Another important feature of the List interface is that it allows **duplicate elements**. The same element can appear multiple times in the collection without causing any errors.*

*Each element in the list is associated with an **index value** that indicates its position. The index of the first element starts at **0**, and the index increases sequentially for each element added to the list. This indexing mechanism allows programmers to easily access and modify elements using their positions.*

For example, a list of numbers may appear as follows:

[10, 20, 30, 20, 40]

*In this list, the number **20 appears twice**, demonstrating that lists allow duplicate values.*

Characteristics of List Interface

The List interface has several important characteristics that distinguish it from other collection interfaces.

*First, it represents an **ordered collection**, meaning that elements are stored in a specific sequence. The order in which elements are inserted into the list is preserved.*

*Second, the List interface allows **duplicate elements**. This means the same value can appear multiple times within the collection.*

*Third, each element in the list has a specific **index position**, which allows elements to be accessed and modified directly using their index values.*

*Another important characteristic is that lists are **dynamic in size**. Unlike arrays, which have a fixed size, lists can automatically grow or shrink when elements are added or removed.*

*Finally, the List interface supports **element traversal**, allowing programmers to iterate through the elements sequentially using loops or iterators.*

Important Methods of List Interface

The List interface defines several methods that are used to perform operations on list elements.

*The **add()** method is used to insert a new element into the list.*

*The **get()** method retrieves the element located at a specific index.*

*The **set()** method replaces an existing element at a particular position with another element.*

*The **remove()** method deletes an element from the list.*

*The **size()** method returns the total number of elements present in the list.*

These methods allow programmers to manage and manipulate list elements efficiently.

Example of List Interface

```
import java.util.ArrayList;
import java.util.List;

public class ListExample
{
   public static void main(String[] args)
   {
      List<String> names = new ArrayList<>();

      names.add("Ravi");
      names.add("Anita");
      names.add("Rahul");
      names.add("Anita");

      System.out.println("List of Names: " + names);
      System.out.println("Element at index 2: " + names.get(2));
      System.out.println("Total elements: " + names.size());
   }
}
```

Explanation

*In this program, a list of names is created using the **ArrayList** class, which implements the List interface. Elements are added to the list using the add() method. The program then retrieves an element using the get() method and displays the number of elements using the size() method.*

*The example also shows that duplicate elements are allowed, since the name **Anita appears twice in the list**.*

2.3.1 ArrayList

Introduction

ArrayList *is one of the most widely used classes in the Java Collections Framework. It is a dynamic implementation of the* ***java.util.List*** *and is available in the* ***java.util****. The ArrayList class provides a flexible way to store and manage collections of objects in an ordered form. In traditional arrays, the size must be defined at the time of creation and cannot be changed later. This limitation makes arrays less flexible for applications where the number of elements may change during program execution. ArrayList overcomes this limitation by allowing the size of the collection to* ***expand or shrink dynamically*** *as elements are added or removed.*

ArrayList maintains the order in which elements are inserted and allows duplicate values. Because of its flexibility and efficiency, ArrayList is commonly used in Java applications for managing lists of data.

Concept of ArrayList

Conceptually, an ***ArrayList*** *can be understood as a* ***resizable array that stores elements in sequential order****. It behaves similarly to an array but provides additional features that make it more powerful and flexible.*

When elements are inserted into an ArrayList, they are stored sequentially. If the number of elements exceeds the current capacity of the internal array, the ArrayList automatically creates a larger array and copies the existing elements into the new array. This automatic resizing ensures that the collection can accommodate more elements without requiring manual intervention.

Each element stored in an ArrayList is associated with an ***index value****, which represents its position in the list. The index of the first element starts at* ***0****, and the index increases sequentially for each additional element. This indexing mechanism allows direct access to elements, making retrieval operations very efficient.*

For example, consider the following ArrayList:

[Java, Python, C++, JavaScript]

The elements are associated with index positions as follows:

Java → index 0
Python → index 1
C++ → index 2
JavaScript → index 3

Using these index positions, elements can be easily accessed or modified.

Internal Working of ArrayList

*Internally, ArrayList uses a **dynamic array data structure** to store elements. Initially, a default capacity is allocated to store elements. When the number of elements increases beyond the current capacity, the ArrayList automatically increases its storage capacity.*
The process generally works as follows:

1. *An internal array is created to store elements.*
2. *Elements are inserted sequentially into the array.*
3. *When the array becomes full, a new larger array is created.*
4. *Existing elements are copied to the new array.*
5. *The new element is then inserted.*

*This mechanism allows ArrayList to provide both **flexibility and efficiency** when handling collections of objects.*

Characteristics of ArrayList

ArrayList has several important characteristics that distinguish it from other collection classes.

Dynamic Size

The size of an ArrayList can change automatically as elements are added or removed. This makes it more flexible than traditional arrays.

Maintains Insertion Order

ArrayList preserves the order in which elements are inserted. When elements are retrieved, they appear in the same order in which they were added.

Allows Duplicate Elements

ArrayList allows duplicate values to be stored. This means the same element can appear multiple times in the list.

Index-Based Access

Each element in an ArrayList can be accessed using its index position. This allows fast retrieval and modification of elements.

Supports Null Values

ArrayList can store null elements along with other objects.

Non-Synchronized Structure

ArrayList is not synchronized by default, which means it is not thread-safe. However, this also makes it faster for single-threaded applications.

Important Methods of ArrayList

ArrayList provides several methods that allow programmers to manipulate and manage elements stored in the collection.
*The **add()** method is used to insert a new element into the list.*
*The **get()** method retrieves the element located at a specified index.*
*The **set()** method replaces an existing element with a new element at a given index.*
*The **remove()** method deletes an element from the list.*
*The **size()** method returns the total number of elements stored in the ArrayList.*
*The **clear()** method removes all elements from the list.*

These methods provide the essential operations required for managing dynamic collections of data.

Example of ArrayList

```
import java.util.ArrayList;

public class ArrayListExample
{
   public static void main(String[] args)
   {
      ArrayList<String> languages = new ArrayList<>();

      languages.add("Java");
      languages.add("Python");
      languages.add("C++");
      languages.add("Java");

      System.out.println("Programming Languages: " + languages);
      System.out.println("Element at index 1: " + languages.get(1));
      System.out.println("Total elements: " + languages.size());

      languages.remove("C++");

      System.out.println("After removing C++: " + languages);
   }
}
```

Explanation

*In this program, an ArrayList named **languages** is created to store programming language names. Elements are added using the add() method. The get() method retrieves an element from a specified index, while the size() method returns the number of elements stored in the list.*

*The program also demonstrates that duplicate values are allowed, as the element **"Java"** **appears twice** in the list. When the remove() method is used, the specified element is removed from the list.*

Advantages of ArrayList

ArrayList offers several advantages that make it widely used in Java programming.

- *It provides **dynamic storage**, allowing the list to grow or shrink automatically.*
- *It supports **fast retrieval of elements using indexes**.*
- *It maintains the **order of elements**.*
- *It allows **duplicate values**.*
- *It simplifies the management of collections compared to traditional arrays.*

Limitations of ArrayList

Despite its advantages, ArrayList also has some limitations.

- *Inserting or deleting elements in the middle of the list can be slower because other elements must be shifted.*
- *It is not synchronized by default, so it may not be suitable for multithreaded environments without additional synchronization.*

2.3.2 LinkedList

Introduction

***LinkedList** is an important class in the Java Collections Framework that implements the **java.util.List**. It is available in the **java.util** and is used to store collections of elements in a sequential order.*

*Unlike ArrayList, which uses a dynamic array structure, LinkedList stores elements using a **linked list data structure**. In this structure, elements are stored in nodes that are connected to each other through links. Each node contains the data and a reference to the next node in the sequence.*

*LinkedList maintains the order of insertion and allows duplicate elements. It is particularly useful when elements need to be **frequently inserted or removed** from the collection.*

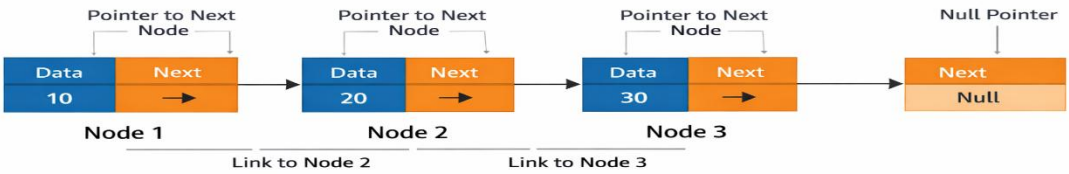
Concept of LinkedList

*Conceptually, a **LinkedList** is a collection of elements where each element is stored in a **separate node**, and each node is connected to the next node through a reference or link.*

A node in a linked list typically contains two parts:

1. ***Data** – the actual element stored in the node*
2. ***Reference (Link)** – a pointer that refers to the next node in the list*

The structure of a linked list can be represented as follows:



In this structure:

- *The first node contains the value **10** and a reference to the next node.*
- *The second node contains the value **20** and points to the third node.*
- *The third node contains the value **30**.*

*Unlike arrays, the elements in a LinkedList are **not stored in contiguous memory locations**. Instead, they are connected through references.*

Internal Working of LinkedList

Internally, a LinkedList works by creating ***nodes*** *that store data and links.*

The process can be explained as follows:

1. *When a new element is inserted, a* ***new node*** *is created.*
2. *The node stores the element as its data.*
3. *The node contains a reference to the next node.*
4. *The previous node is updated to point to the new node.*

Because of this structure, inserting or deleting elements does not require shifting other elements as in arrays. Instead, only the references between nodes need to be updated.

Characteristics of LinkedList

LinkedList has several important characteristics.

Maintains Insertion Order

Elements in a LinkedList are stored in the order in which they are inserted.

Allows Duplicate Elements

LinkedList allows duplicate values to be stored in the collection.

Dynamic Data Structure

The size of the LinkedList can grow or shrink dynamically as elements are added or removed.

Efficient Insertion and Deletion

LinkedList is efficient when inserting or deleting elements because it does not require shifting other elements.

Sequential Access

Elements are accessed sequentially rather than directly using indexes.

Supports Null Elements

LinkedList can store null values along with other elements.

Important Methods of LinkedList

LinkedList provides several methods to manipulate elements in the collection.

add()

Adds an element to the list.

get()

Retrieves the element at a specific index.

remove()

Removes an element from the list.

size()

Returns the number of elements in the list.

addFirst()

Adds an element at the beginning of the list.

addLast()

Adds an element at the end of the list.

Example of LinkedList

```
import java.util.LinkedList;

public class LinkedListExample
{
   public static void main(String[] args)
   {
      LinkedList<String> cities = new LinkedList<>();

      cities.add("Delhi");
      cities.add("Mumbai");
      cities.add("Hyderabad");
      cities.add("Delhi");

      System.out.println("Cities List: " + cities);

      cities.addFirst("Chennai");

      System.out.println("After adding at first: " + cities);

      cities.remove("Mumbai");

      System.out.println("After removing Mumbai: " + cities);
   }
}
```

Explanation

*In this example, a **LinkedList** named cities is created to store city names. Elements are added using the add() method. The addFirst() method inserts an element at the beginning of the list, while the remove() method deletes an element from the list.*

*The program also shows that duplicate values are allowed because the city **"Delhi" appears twice** in the list.*

Advantages of LinkedList

LinkedList provides several advantages in Java programming.

- *Efficient insertion and deletion of elements*
- *Dynamic size that can grow or shrink automatically*
- *Maintains insertion order of elements*
- *Allows duplicate values*

Limitations of LinkedList

Despite its advantages, LinkedList also has some limitations.

- *Accessing elements can be slower because traversal must occur sequentially.*
- *It requires more memory because each node stores an additional reference.*

2.4 Set Interface

Introduction

*The **Set Interface** is an important part of the Java Collections Framework that represents a **collection of unique elements**. Unlike lists, sets do not allow duplicate values. If an attempt is made to insert a duplicate element, the set simply ignores it and retains only one instance of that element.*

*The Set interface extends the **java.util.Collection** and is available in the **java.util**. It defines the basic operations required to store and manipulate collections of distinct objects.*

*Several classes implement the Set interface, including **java.util.HashSet**, **java.util.TreeSet**, and LinkedHashSet. These classes provide different internal mechanisms for storing elements while ensuring that duplicates are not allowed.*

Concept of Set Interface

*Conceptually, the Set interface represents a **collection that contains only unique elements**. The primary objective of a set is to prevent duplicate values from being stored in the collection.*

When an element is added to a set, the collection checks whether the element already exists. If the element is already present, the new element is not added. This property ensures that each element in the set remains distinct.

For example, consider the following set of numbers:

{10, 20, 30, 40}

*If an attempt is made to insert the value **20** again, the set will ignore the duplicate value, and the collection will remain unchanged.*

Unlike lists, sets generally do not maintain a fixed order of elements. The order in which elements appear may vary depending on the implementation of the set.

Characteristics of Set Interface

The Set interface has several distinctive characteristics.

Unique Elements

A set does not allow duplicate elements. Each value must be unique within the collection.

No Index-Based Access

Elements in a set cannot be accessed using index positions because sets do not maintain element positions like lists.

Dynamic Collection

The size of a set can change dynamically as elements are added or removed.

Part of Collection Hierarchy

The Set interface extends the Collection interface and inherits its basic operations.

Supports Iteration

Elements stored in a set can be traversed using iterators or loops.

Important Methods of Set Interface

The Set interface inherits many methods from the Collection interface that allow operations on elements.

*The **add()** method is used to insert an element into the set. Duplicate elements are automatically ignored.*

*The **remove()** method removes a specified element from the set.*

*The **contains()** method checks whether a specific element exists in the set.*

*The **size()** method returns the total number of elements present in the set.*

*The **clear()** method removes all elements from the set.*

Example of Set Interface

```
import java.util.HashSet;
import java.util.Set;

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

```
    {
        Set<String> languages = new HashSet<>();

        languages.add("Java");
        languages.add("Python");
        languages.add("C++");
        languages.add("Java");

        System.out.println("Set Elements: " + languages);
    }
}
```

Explanation

*In this example, a set of programming languages is created using the **HashSet** class. Elements are inserted using the add() method. Although the element **"Java"** is inserted twice, it appears only once in the set because sets do not allow duplicate elements.*

2.7 HashSet

Introduction

***HashSet** is one of the most commonly used classes in the Java Collections Framework. It implements the **java.util.Set** and is available in the **java.util**. HashSet is designed to store a collection of **unique elements**, meaning that duplicate values are not allowed.*

*The main feature of HashSet is that it uses a **hash table data structure** internally to store elements. Because of this structure, HashSet provides very fast performance for operations such as insertion, deletion, and searching.*

HashSet does not maintain the order of elements. When elements are displayed, they may appear in a different order from the order in which they were inserted.

Concept of HashSet

*Conceptually, a **HashSet** can be understood as a collection that stores elements in a **hash table**. A hash table is a data structure that stores data using a special function called a **hash function**.*

*When an element is inserted into a HashSet, the hash function converts the element into a **hash code**, which determines the position where the element will be stored in memory. This process allows the HashSet to locate elements very quickly.*

Since the Set interface does not allow duplicate elements, HashSet automatically checks whether an element already exists before inserting it. If the element is already present, the new element is ignored.

For example:

{10, 20, 30, 40}

*If the value **20** is inserted again, it will not be added because HashSet stores only unique elements.*

Characteristics of HashSet

HashSet has several important characteristics.

Unique Elements

HashSet does not allow duplicate elements. Each element in the set must be unique.

No Guaranteed Order

HashSet does not maintain the insertion order of elements. The order of elements may appear random.

Fast Performance

HashSet provides fast operations for adding, removing, and searching elements because it uses hashing.

Allows Null Values

*HashSet allows a single **null** element to be stored.*

Non-Synchronized

HashSet is not synchronized by default, meaning it is not thread-safe in multithreaded environments.

Important Methods of HashSet

HashSet provides several methods that allow manipulation of elements stored in the set.

add()

Adds an element to the HashSet.

remove()

Removes an element from the set.

contains()

Checks whether a particular element exists in the set.

size()

Returns the number of elements present in the set.

clear()

Removes all elements from the set.

Example of HashSet

```
import java.util.HashSet;

public class HashSetExample
{
   public static void main(String[] args)
   {
      HashSet<String> fruits = new HashSet<>();

      fruits.add("Apple");
      fruits.add("Mango");
      fruits.add("Orange");
      fruits.add("Apple");
```

```
        System.out.println("HashSet Elements: " + fruits);
        System.out.println("Total elements: " + fruits.size());
    }
}
```

Explanation

*In this example, a HashSet named **fruits** is created to store fruit names. Elements are added using the add() method. Although the element **"Apple"** is inserted twice, it appears only once in the output because HashSet does not allow duplicate elements.*
The size() method returns the number of unique elements present in the set.

Advantages of HashSet

HashSet provides several advantages:

- *Stores **unique elements only***
- *Provides **fast insertion and searching operations***
- *Allows dynamic growth of the collection*
- *Simple and efficient for storing unordered data*

Limitations of HashSet

Despite its advantages, HashSet also has some limitations:

- *Does not maintain insertion order*
- *Not synchronized by default*
- *Does not support index-based access*

2.8 TreeSet

Introduction

***TreeSet** is an important class in the Java Collections Framework used to store and manage a collection of **unique elements in sorted order**. It implements the **java.util.Set** and is available in the **java.util**. Because it is a Set implementation, TreeSet does not allow duplicate values. The distinguishing feature of TreeSet is that it **automatically maintains elements in a sorted sequence**. Whenever elements are inserted into the collection, TreeSet arranges them according to their natural ordering. This means the smallest element appears first and the largest element appears last.*
Unlike some other collection classes that simply store elements without organization, TreeSet ensures that the stored elements remain ordered at all times. This makes it particularly useful in situations where data must be maintained in sorted form.

Concept of TreeSet

*Conceptually, a **TreeSet** can be understood as a collection that organizes elements using a **tree-based data structure**. Internally, it uses a self-balancing binary search tree (commonly known as a **Red-Black Tree**) to store elements.*
In a tree structure, elements are arranged hierarchically. Each element is compared with other elements to determine its correct position in the structure. When a new element is

added, the collection performs comparisons with existing elements and places the new element in the appropriate location.
Because the tree remains balanced and ordered, the elements of a TreeSet always appear in ***sorted order*** *when displayed.*
For example, if the following elements are inserted:
40, 10, 30, 20
TreeSet will automatically arrange them as:
[10, 20, 30, 40]
Even though the elements were inserted in an unsorted sequence, TreeSet reorganizes them internally so that the output appears sorted.

Internal Working of TreeSet
The internal working of TreeSet is based on a ***balanced tree structure****. When elements are added to the TreeSet, the following steps occur:*

1. *The new element is compared with existing elements in the tree.*
2. *Based on the comparison result, the element is placed in the correct position within the tree.*
3. *The tree structure automatically adjusts itself to maintain balance.*
4. *As a result, the tree remains organized and efficient for searching, insertion, and deletion operations.*

Because the elements are stored in a sorted tree structure, operations such as searching, inserting, and deleting elements can be performed efficiently.

Characteristics of TreeSet
TreeSet has several important characteristics that distinguish it from other collection classes.
Sorted Elements
TreeSet automatically arranges elements in ascending order according to their natural ordering.
Unique Elements
Like all Set implementations, TreeSet does not allow duplicate values. Each element must be unique.
Tree-Based Storage
TreeSet uses a balanced tree structure to store elements, which ensures efficient ordering and retrieval.
No Index-Based Access
Unlike lists, TreeSet does not provide index-based access because elements are not stored according to positions but according to their sorted order.
Dynamic Collection
TreeSet automatically grows or shrinks as elements are added or removed.
Does Not Allow Null Values
TreeSet generally does not allow null elements because comparison operations cannot be performed on null values.

Important Methods of TreeSet

TreeSet provides several methods for managing elements in the collection.

add()

This method inserts an element into the TreeSet while maintaining sorted order.

remove()

Removes a specified element from the collection.

contains()

Checks whether a particular element exists in the TreeSet.

first()

Returns the smallest element present in the set.

last()

Returns the largest element in the set.

size()

Returns the total number of elements in the TreeSet.

clear()

Removes all elements from the collection.

Example of TreeSet

```
import java.util.TreeSet;

public class TreeSetExample
{
   public static void main(String[] args)
   {
      TreeSet<Integer> numbers = new TreeSet<>();

      numbers.add(40);
      numbers.add(10);
      numbers.add(30);
      numbers.add(20);

      System.out.println("TreeSet Elements: " + numbers);

      System.out.println("Smallest Element: " + numbers.first());
      System.out.println("Largest Element: " + numbers.last());
   }
}
```

Explanation

In this example, a TreeSet named ***numbers*** *is created to store integer values. The elements are added using the add() method.*

Although the numbers are inserted in the order ***40, 10, 30, 20****, the output appears as:*

[10, 20, 30, 40]

This occurs because TreeSet automatically sorts the elements according to their natural order. The methods first() and last() are used to retrieve the smallest and largest elements respectively.

Advantages of TreeSet

TreeSet provides several advantages in Java programming.

- *Automatically maintains elements in **sorted order***
- *Ensures that all elements are **unique***
- *Provides efficient searching and retrieval operations*
- *Useful for applications that require **ordered data***

Limitations of TreeSet

Despite its advantages, TreeSet also has certain limitations.

- *It does not maintain the **insertion order** of elements.*
- *Accessing elements can be slower compared to HashSet because sorting and comparisons are required.*
- *It does not support **index-based access**.*

2.9 Iterator

Introduction

*The **Iterator** is an important interface in the Java Collections Framework that provides a mechanism for **accessing elements of a collection sequentially**. It is mainly used to traverse elements stored in collection objects without exposing the internal structure of the collection. The Iterator interface is part of the **java.util** and is commonly used with collection classes such as **java.util.ArrayList**, **java.util.LinkedList**, **java.util.HashSet**, and **java.util.TreeSet**. In many applications, collections may contain a large number of elements. Accessing these elements one by one requires a systematic mechanism. The Iterator interface provides such a mechanism by allowing programs to **move through elements of a collection step by step**.*

Concept of Iterator

*Conceptually, an **Iterator** can be understood as a **cursor or pointer that moves through the elements of a collection**. Instead of accessing elements randomly, the iterator visits each element sequentially.*

*When an iterator is created, it initially points **before the first element** of the collection. As the iterator moves forward, it accesses each element in sequence.*

For example, consider the following collection:

[Java, Python, C++, JavaScript]

The iterator will traverse the elements in the following order:

Java → Python → C++ → JavaScript

*Thus, the iterator acts as a **navigation tool** that allows a program to move through a collection and process each element individually.*

Internal Working of Iterator

*The Iterator interface works by maintaining a **position within the collection**. The traversal process generally follows these steps:*

1. *A collection object is created and elements are inserted.*
2. *An iterator object is obtained from the collection using the iterator() method.*
3. *The iterator checks whether another element exists using the hasNext() method.*
4. *If another element exists, the next() method retrieves the element and moves the iterator forward.*
5. *This process continues until all elements have been accessed.*

*This mechanism ensures that elements are accessed **in a controlled and sequential manner**.*

Characteristics of Iterator

The Iterator interface has several important characteristics that make it useful in collection traversal.

Sequential Access

Iterator allows elements to be accessed one at a time in sequence rather than randomly.

Universal Traversal Mechanism

Iterator provides a common way to traverse different types of collections such as lists and sets.

Independent of Internal Structure

Iterator allows traversal without revealing how the collection stores its elements internally.

Safe Element Removal

Iterator provides a method that allows elements to be removed safely while iterating through the collection.

Simplifies Collection Navigation

Iterator eliminates the need for complex looping mechanisms and provides a simple method for navigating through collections.

Important Methods of Iterator

The Iterator interface defines three primary methods that control the traversal process.

hasNext()

*This method checks whether there are more elements remaining in the collection. It returns **true** if another element is available.*

next()

This method retrieves the next element in the collection and moves the iterator forward to the next position.

remove()

This method removes the last element returned by the iterator from the collection.

Iterator Traversal Process

The working of an iterator can be represented conceptually as follows:

Collection Elements

[10, 20, 30, 40]

Iterator Movement

Start → 10 → 20 → 30 → 40 → End
At each step, the iterator retrieves the current element and moves to the next element.

Example of Iterator

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample
{
   public static void main(String[] args)
   {
      ArrayList<String> languages = new ArrayList<>();

      languages.add("Java");
      languages.add("Python");
      languages.add("C++");
      languages.add("JavaScript");

      Iterator<String> itr = languages.iterator();

      while(itr.hasNext())
      {
         String lang = itr.next();
         System.out.println(lang);
      }
   }
}
```

Explanation

*In this example, an **ArrayList** named languages is created to store programming languages.*
An iterator object is obtained using the iterator() method.
The hasNext() method checks whether another element exists in the collection. If it returns true, the next() method retrieves the next element.
This process continues until the iterator reaches the end of the collection.

Advantages of Iterator

Iterator provides several advantages when working with collections.

- *Provides a **standardized method for traversing collections***
- *Allows **safe removal of elements during traversal***
- *Works with different types of collections*
- *Simplifies the process of accessing collection elements*

Limitations of Iterator

Despite its advantages, Iterator also has some limitations.

- *It supports only **forward traversal** of elements.*
- *It does not allow direct access to elements using index positions.*
- *It may be slower for very large collections when compared to indexed access.*

2.6 Map Interface

Introduction

*The **Map Interface** is an important component of the Java Collections Framework that is used to store data in the form of **key–value pairs**. In many applications, information is often stored in pairs where one value is used to identify another value. The Map interface provides a convenient structure to manage such relationships.*

*The Map interface is available in the **java.util**. Unlike other collection interfaces such as **java.util.List** or **java.util.Set**, the Map interface does not extend the **java.util.Collection**. This is because it stores data as **pairs of keys and values**, rather than storing individual elements. Several classes implement the Map interface, such as **java.util.HashMap**, **java.util.TreeMap**, and LinkedHashMap. These classes differ in the way they store and organize key–value pairs.*

Concept of Map Interface

*Conceptually, a **Map** can be understood as a collection that **associates one object with another object**. In this structure, each element consists of two parts:*

- ***Key** – an identifier used to locate data*
- ***Value** – the actual data associated with the key*

The key acts as a reference that allows the program to quickly retrieve the corresponding value.

*A very common real-world example of a map is a **dictionary**. In a dictionary, a word acts as the key and its meaning acts as the value. By using the word, we can easily find its meaning.*

For example:

Key (Word) Value (Meaning)

Java Programming Language

CPU Central Processing Unit

RAM Random Access Memory

In the same way, a Java Map stores data in key–value pairs so that values can be accessed quickly using their keys.

Internal Idea of Map Storage

*When data is stored in a Map, each key is linked to a specific value. The Map ensures that **each key is unique**, meaning that no two keys can be the same.*

If an attempt is made to insert a new value using an existing key, the previous value associated with that key will be replaced by the new value.

For example:

101 → Ravi
102 → Anita
103 → Rahul
Here:

- ***101, 102, 103** are keys*
- ***Ravi, Anita, Rahul** are values*

If the following operation is performed:
101 → Kiran
*The value **Ravi** will be replaced by **Kiran** because the key **101** already exists.*

Characteristics of Map Interface
The Map interface has several important characteristics.
Key–Value Pair Structure
A Map stores elements as pairs of keys and values rather than as single objects.
Unique Keys
Each key must be unique. Duplicate keys are not allowed in a Map.
Duplicate Values Allowed
Although keys must be unique, values may be repeated.
No Index-Based Access
Unlike lists, Map elements cannot be accessed using index positions. Instead, values are retrieved using their corresponding keys.
Efficient Data Retrieval
Maps allow fast retrieval of data because values can be accessed directly using their keys.
Dynamic Collection
The size of a Map can change dynamically as elements are added or removed.

Important Methods of Map Interface
The Map interface provides several methods for managing key–value pairs.
put()
The put() method is used to insert a key–value pair into the map.
Example:
map.put(101, "Ravi");

---

get()
The get() method retrieves the value associated with a specific key.
Example:
map.get(101);

---

remove()
This method removes a key–value pair from the map.

---

containsKey()
Checks whether a particular key exists in the map.

containsValue()
Checks whether a specific value exists in the map.

size()
Returns the total number of key–value pairs stored in the map.

clear()
Removes all entries from the map.

Example of Map Interface

```
import java.util.HashMap;

public class MapExample
{
   public static void main(String[] args)
   {
      HashMap<Integer, String> students = new HashMap<>();

      students.put(101, "Ravi");
      students.put(102, "Anita");
      students.put(103, "Rahul");

      System.out.println("Student with ID 101: " + students.get(101));
      System.out.println("All Students: " + students);
   }
}
```

Explanation
*In this program, a **HashMap** named students is created to store student IDs and names. The put() method inserts key–value pairs into the map. Each student ID acts as a key and the corresponding student name acts as the value.*
*The get() method retrieves the value associated with a specific key. When the key **101** is used, the program returns the value **Ravi**.*

Map Interface Hierarchy
The structure of the Map interface and its common implementations can be represented as follows:

- *Map Interface*
  HashMap
  TreeMap

HashMap

HashMap *stores key–value pairs using a* ***hash table mechanism****. It provides fast insertion and retrieval operations but does not maintain any specific order of elements.*

TreeMap

TreeMap *stores key–value pairs in* ***sorted order based on keys****. It uses a tree-based data structure internally and automatically arranges keys in ascending order.*

Advantages of Map Interface

The Map interface provides several advantages.

- *Allows efficient storage of* ***key–value relationships***
- *Enables* ***fast retrieval of values using keys***
- *Prevents duplication of keys*
- *Useful for representing structured data*

3. Utility Classes

Introduction

In Java programming, ***Utility Classes*** *are special classes that provide methods to perform common operations required in programs. These classes simplify programming by offering ready-made functionalities for tasks such as* ***string processing, date and time management, calendar calculations, and input handling****.*

Utility classes help programmers avoid writing lengthy or repetitive code for frequently used operations. Instead of creating new methods for basic tasks, developers can use the built-in methods provided by Java. This approach improves ***program readability, efficiency, and development speed****.*

Many utility classes are available in the ***java.util****, which contains several useful classes designed to support everyday programming activities such as handling strings, managing dates, and reading user input.*

Conceptually, a utility class can be viewed as a ***supporting class that provides useful tools for solving common programming problems****. These classes usually contain methods that perform specific operations and assist other parts of the program. Unlike regular classes that represent real-world objects, utility classes mainly focus on providing* ***helpful methods and functionalities****.*

Some commonly used utility classes in Java include:

- ***java.util.StringTokenizer*** *– used to divide a string into smaller components called tokens.*
- ***java.util.Date*** *– used to represent and manipulate date and time information.*
- ***java.util.Calendar*** *– used to perform various date and time calculations.*
- ***java.util.GregorianCalendar*** *– a subclass of Calendar that represents the widely used Gregorian calendar system.*
- ***java.util.Scanner*** *– used to read input data from the keyboard or other input sources.*

Each of these utility classes provides specialized methods designed to perform particular tasks. By using these classes, programmers can easily perform operations such as ***splitting strings, managing time-related data, and reading user input*** *in an efficient and organized manner.*

Thus, utility classes play an important role in Java programming because they provide ***powerful and reusable tools that simplify program development and improve the overall structure of applications****.*

3.1 StringTokenizer

Introduction

The ***java.util.StringTokenizer*** *is a utility class in Java used to* ***break a string into smaller parts called tokens****. A token represents a piece of text separated by a specific character known as a* ***delimiter****.*

This class is available in the ***java.util*** *and is commonly used when a program needs to process individual words or elements from a larger string.*

In many applications, data is stored in the form of long strings that contain multiple values separated by spaces, commas, or other characters. The StringTokenizer class helps in ***separating these values into manageable components*** *so that each element can be processed individually.*

Concept of StringTokenizer

Conceptually, ***StringTokenizer works by dividing a string into tokens based on a delimiter****.*

A delimiter is a character that marks the boundary between different tokens.

For example, consider the following string:

Java Python C++ JavaScript

If space is used as the delimiter, the string will be divided into the following tokens:

Java

Python

C++

JavaScript

Each token represents a separate word that can be processed independently by the program. StringTokenizer allows the program to ***read and extract tokens one by one*** *until the entire string has been processed.*

Working of StringTokenizer

The working process of StringTokenizer generally involves the following steps:

1. *A string containing multiple values is provided.*
2. *A delimiter is specified to indicate how the string should be split.*
3. *The StringTokenizer object separates the string into tokens.*
4. *Each token is retrieved sequentially until no tokens remain.*

This mechanism allows programs to handle text data efficiently.

Important Methods of StringTokenizer

The StringTokenizer class provides several methods that help in tokenizing strings.

hasMoreTokens()

This method checks whether additional tokens are available in the string.

nextToken()

This method returns the next token from the string.

countTokens()

This method returns the total number of tokens remaining in the string.
These methods allow programmers to navigate through tokens easily.

Example of StringTokenizer

```
import java.util.StringTokenizer;

public class TokenExample
{
   public static void main(String[] args)
   {
      String text = "Java Python C++ JavaScript";

      StringTokenizer st = new StringTokenizer(text);

      while(st.hasMoreTokens())
      {
         System.out.println(st.nextToken());
      }
   }
}
```

Explanation

In this program, a string containing the names of programming languages is created. A ***StringTokenizer*** *object is then used to divide the string into tokens based on spaces.*
The hasMoreTokens() method checks whether more tokens are available, and the nextToken() method retrieves each token one by one until all tokens have been displayed.

Advantages of StringTokenizer

StringTokenizer provides several benefits in Java programming.

- *It simplifies the process of* ***splitting strings into smaller parts****.*
- *It allows sequential access to tokens.*
- *It helps process large text strings efficiently.*
- *It reduces the need for manual string parsing.*

Limitations of StringTokenizer

Although StringTokenizer is useful, it also has certain limitations.

- *It provides fewer features compared to newer string-processing classes.*
- *Modern Java programs often use the split() method of the String class instead.*

3.2 Date

Introduction

*The **java.util.Date** class is one of the important utility classes in Java used to represent **date and time information**. It provides a simple way for Java programs to work with time-related data such as the current date, current time, and specific time instances.*
*This class is available in the **java.util** and is widely used in applications that need to record or display time-related events. For example, systems such as banking applications, logging systems, ticket booking platforms, and scheduling software often need to store and process date and time information.*
*The Date class represents **a specific instant in time**, which includes details such as the day, month, year, hour, minute, and second. It allows programmers to create objects that represent particular moments and perform operations related to time.*

Concept of Date Class

*Conceptually, the Date class represents **a single point in time measured in milliseconds**.*
*Internally, Java stores the time as the number of milliseconds that have passed since a specific reference date called the **epoch**.*
The epoch time in Java is defined as:
January 1, 1970, 00:00:00 GMT
Every Date object internally contains a numeric value that indicates the number of milliseconds that have elapsed since this reference point. Using this value, Java can determine the exact date and time.
For example, if a Date object stores the value representing the current system time, Java converts that value into a human-readable format such as:
Wed Mar 11 11:30:15 IST 2026
This representation includes the day of the week, month, day of the month, time, time zone, and year.
*Thus, the Date class acts as a **bridge between machine-readable time values and human-readable date formats**.*

Internal Representation of Date

*Internally, the Date class uses a **long integer value** to represent time. This value indicates the number of milliseconds that have passed since the epoch.*
For example:
Epoch Time = January 1, 1970, 00:00:00 GMT
If the current time corresponds to a certain number of milliseconds after the epoch, that value is stored inside the Date object.
Conceptually:
Date Object

```
    |
    └── Time in milliseconds since 1970
```

Java then converts this millisecond value into a readable format when the date is printed.

Characteristics of Date Class

The Date class has several important characteristics that define its behavior.

Represents a Specific Moment in Time
A Date object represents a particular instant in time rather than just a calendar date.
Millisecond-Based Representation
Time is internally stored as the number of milliseconds elapsed since the epoch.
Supports Date Comparison
Date objects can be compared with one another to determine which date occurs earlier or later.
Supports Conversion to Readable Format
The Date class converts internal time values into readable date and time strings.
Part of Java Utility Library
The Date class is included in the java.util package and forms part of Java's utility library.

Constructors of Date Class
The Date class provides constructors that allow objects to be created in different ways.
Default Constructor
This constructor creates a Date object representing the ***current system date and time****.*
Example:
Date d = new Date();
Parameterized Constructor
This constructor allows a Date object to be created using a specific millisecond value.
Example:
Date d = new Date(1000000);
This creates a Date object representing a time based on the specified milliseconds after the epoch.

Important Methods of Date Class
The Date class provides several useful methods for working with time values.
toString()
Returns the date and time in a human-readable string format.
getTime()
Returns the number of milliseconds since January 1, 1970.
before()
Checks whether one Date occurs before another Date.
after()
Checks whether one Date occurs after another Date.
equals()
Checks whether two Date objects represent the same moment in time.

Example of Date Class

```
import java.util.Date;

public class DateExample
{
```

```
    public static void main(String[] args)
    {
        Date currentDate = new Date();

        System.out.println("Current Date and Time: " + currentDate);

        long milliseconds = currentDate.getTime();
        System.out.println("Milliseconds since epoch: " + milliseconds);
    }
}
```

Explanation

In this example, a Date object named currentDate is created using the default constructor. Since no parameters are provided, the object automatically stores the ***current system date and time****.*

When the object is printed, the toString() method converts the stored millisecond value into a readable date format.

The getTime() method returns the internal millisecond value representing the time since January 1, 1970.

Advantages of Date Class

The Date class provides several benefits for handling time-related information.

- *It allows programs to easily obtain the* ***current date and time****.*
- *It provides methods for* ***comparing different dates****.*
- *It supports conversion between* ***millisecond values and readable date formats****.*
- *It helps record time-related events in applications.*

Limitations of Date Class

Although the Date class is useful, it has some limitations.

- *Some of its older methods are deprecated.*
- *It provides limited support for complex date manipulation.*
- *More advanced date handling is usually performed using the Calendar class or newer Java time libraries.*

3.3 Calendar

Introduction

The ***java.util.Calendar*** *is an important utility class in Java used to* ***represent and manipulate date and time information****. It provides methods that allow programmers to retrieve and modify different components of a date such as the year, month, day, hour, minute, and second.*

The Calendar class is part of the ***java.util*** *and is more powerful and flexible than the Date class. While the Date class mainly represents a specific moment in time, the Calendar class allows programmers to* ***perform calculations and adjustments on individual date fields****.*

This class is widely used in applications that require operations such as scheduling events, calculating future or past dates, and retrieving specific date components.

Concept of Calendar Class

*Conceptually, the Calendar class can be understood as a **system that converts a specific moment in time into meaningful calendar fields**. Instead of simply storing time as milliseconds, it breaks down time into different components such as:*

- *Year*
- *Month*
- *Day*
- *Hour*
- *Minute*
- *Second*

*These components are called **calendar fields**.*

For example, a calendar date may be represented as:

Year : 2026
Month : March
Day : 11
Hour : 10
Minute : 45
Second : 30

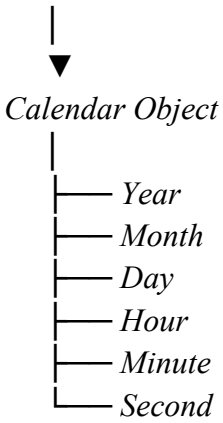
The Calendar class allows programs to access and modify these individual fields easily.

Internal Concept of Calendar

*Internally, the Calendar class represents time using **milliseconds since January 1, 1970**, which is known as the epoch time. However, it converts this time value into human-readable calendar fields.*

Conceptually:

Milliseconds since Epoch



*Thus, the Calendar class acts as an interface between the **numeric representation of time** and **human-readable date components**.*

Characteristics of Calendar Class

The Calendar class has several important characteristics.

Field-Based Date Representation

The Calendar class stores date and time information as separate fields such as year, month, and day.

Date Manipulation

It allows programmers to modify date values by adding or subtracting days, months, or years.

Flexible Time Handling

The class supports multiple calendar systems and time zones.

Abstract Class

Calendar is an abstract class, meaning objects cannot be created directly using the new keyword. Instead, the getInstance() method is used to obtain an object.

Supports Date Calculations

It allows complex calculations such as determining future dates, past dates, and leap years.

Creating a Calendar Object

Since Calendar is an abstract class, an object is created using the ***getInstance()*** *method.*

Example:

```
Calendar cal = Calendar.getInstance();
```

This method returns a Calendar object that represents the ***current system date and time****.*

Important Methods of Calendar Class

The Calendar class provides several methods to access and manipulate date information.

get()

This method retrieves the value of a specific calendar field.

Example:

```
cal.get(Calendar.YEAR);
```

---

set()

This method modifies the value of a particular calendar field.

Example:

```
cal.set(Calendar.YEAR, 2030);
```

---

add()

This method adds or subtracts a specific amount from a calendar field.

Example:

```
cal.add(Calendar.DATE, 5);
```

---

getTime()

Returns the date and time represented by the Calendar object.

Example of Calendar Class

```
import java.util.Calendar;

public class CalendarExample
{
   public static void main(String[] args)
   {
      Calendar cal = Calendar.getInstance();

      int year = cal.get(Calendar.YEAR);
      int month = cal.get(Calendar.MONTH);
      int day = cal.get(Calendar.DATE);

      System.out.println("Year: " + year);
      System.out.println("Month: " + month);
      System.out.println("Day: " + day);
   }
}
```

Explanation

*In this program, a Calendar object is created using the getInstance() method. The program retrieves the current **year, month, and day** using the get() method.*
Each calendar field is accessed using predefined constants such as Calendar.YEAR, Calendar.MONTH, and Calendar.DATE.

Advantages of Calendar Class

The Calendar class offers several advantages.

- *It allows easy **manipulation of date and time values**.*
- *It provides methods for **retrieving individual date fields**.*
- *It supports **date calculations** such as adding days or months.*
- *It handles **different time zones and calendar systems**.*

Limitations of Calendar Class

Despite its advantages, the Calendar class also has some limitations.

- *The syntax can sometimes be complex.*
- *Newer Java versions provide improved date and time classes in the Java Time API.*

3.4 GregorianCalendar

Introduction

*The **java.util.GregorianCalendar** is a class in the Java Collections and Utility library that is used to represent and manipulate **dates and times according to the Gregorian calendar system**. It is a subclass of the **java.util.Calendar**, which means it inherits all the features and methods provided by the Calendar class.*
*The Gregorian calendar is the **most widely used calendar system in the world today**, and it is the standard calendar used for civil purposes in most countries. The GregorianCalendar*

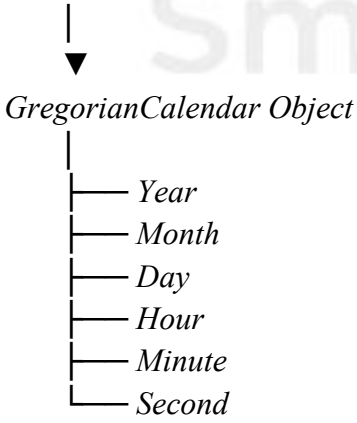
class provides an implementation of the Calendar class that follows the rules of this calendar system.
This class is available in the ***java.util*** *and is used when programs need to perform* ***date calculations, time manipulation, and calendar-based operations****.*

Concept of GregorianCalendar
Conceptually, the GregorianCalendar class represents ***a system for organizing time into meaningful calendar fields such as year, month, and day*** *according to the Gregorian calendar rules.*
The Gregorian calendar introduced improvements over earlier calendar systems by adjusting the calculation of leap years. According to this system:

- *A year is a* ***leap year if it is divisible by 4****.*
- *However, years divisible by* ***100 are not leap years****.*
- *But years divisible by* ***400 are leap years****.*

For example:
2000 → Leap Year
1900 → Not a Leap Year
2024 → Leap Year
The GregorianCalendar class automatically applies these rules when performing date calculations.

Internal Working of GregorianCalendar
Internally, the GregorianCalendar class stores time as ***milliseconds since January 1, 1970****, which is known as the* ***epoch time****. However, it converts this internal value into meaningful calendar fields.*
Conceptually, the process works as follows:
Milliseconds since Epoch



Thus, the class acts as a bridge between the internal representation of time and the human-readable calendar format.

Characteristics of GregorianCalendar
The GregorianCalendar class has several important characteristics.
Inherits Calendar Functionality

Since GregorianCalendar extends the Calendar class, it inherits all its methods and properties.

Supports Date Calculations

It allows programs to perform operations such as adding or subtracting days, months, and years.

Handles Leap Years

The class automatically applies the leap year rules of the Gregorian calendar.

Field-Based Date Representation

Dates are represented using separate fields such as year, month, day, hour, and minute.

Time Zone Support

GregorianCalendar also supports different time zones and localization.

Creating a GregorianCalendar Object

A GregorianCalendar object can be created using its constructor.

Example:

GregorianCalendar gc = new GregorianCalendar();

*This creates a calendar object representing the **current system date and time**.*

It is also possible to specify a particular date.

Example:

GregorianCalendar gc = new GregorianCalendar(2026, 2, 11);

*This creates a calendar object representing **March 11, 2026** (months are indexed starting from zero).*

Important Methods of GregorianCalendar

The GregorianCalendar class provides several methods for working with date and time.

get()

Retrieves the value of a specific calendar field.

Example:

gc.get(Calendar.YEAR);

set()

Sets a particular calendar field to a specified value.

Example:

gc.set(Calendar.YEAR, 2030);

add()

Adds a specified amount of time to a calendar field.

Example:

gc.add(Calendar.DATE, 5);

isLeapYear()

Determines whether a given year is a leap year.

Example:

gc.isLeapYear(2024);

Example of GregorianCalendar

```
import java.util.GregorianCalendar;
import java.util.Calendar;

public class GregorianExample
{
   public static void main(String[] args)
   {
      GregorianCalendar gc = new GregorianCalendar();

      int year = gc.get(Calendar.YEAR);
      int month = gc.get(Calendar.MONTH);
      int day = gc.get(Calendar.DATE);

      System.out.println("Year: " + year);
      System.out.println("Month: " + month);
      System.out.println("Day: " + day);

      System.out.println("Is Leap Year: " + gc.isLeapYear(year));
   }
}
```

Explanation

*In this example, a GregorianCalendar object is created to represent the current date. The program retrieves the **year, month, and day** using the get() method.*
The isLeapYear() method checks whether the current year is a leap year according to the Gregorian calendar rules.

Advantages of GregorianCalendar

The GregorianCalendar class provides several benefits.

- *It follows the **internationally accepted Gregorian calendar system**.*
- *It supports **accurate leap year calculations**.*
- *It allows flexible **date manipulation and arithmetic operations**.*
- *It supports **time zone management**.*

Limitations of GregorianCalendar

Despite its usefulness, the class has certain limitations.

- *Its syntax can sometimes be complex for beginners.*
- *Modern Java applications often use the **Java Time API** for more advanced date handling.*

3.5 Scanner

Introduction

*The **java.util.Scanner** is an important utility class in Java used to **read input from the user or from different input sources**. It is commonly used in Java programs to obtain input from the **keyboard**, but it can also read input from files, strings, and other input streams.*
*The Scanner class is part of the **java.util** and provides an easy and flexible way to read different types of data such as integers, floating-point numbers, and strings.*
*In many programs, user interaction is required. For example, a program may need to ask the user to enter their name, age, or marks. The Scanner class makes it possible to **capture this input and process it within the program**.*

Concept of Scanner Class

*Conceptually, the Scanner class acts as an **input reader and parser**. It reads the input provided by the user and breaks it into smaller pieces called **tokens**. These tokens are then converted into appropriate data types such as integers, strings, or floating-point numbers.*
For example, when a user enters the following input:
25
The Scanner object reads this input and converts it into an integer value that can be used in the program.
Similarly, if a user enters text such as:
Java Programming
The Scanner class can read it as a string.
*Thus, the Scanner class serves as a **bridge between user input and program processing**.*

Working of Scanner Class

The Scanner class works through the following steps:

1. *A Scanner object is created to read input.*
2. *The input source is specified (usually the keyboard).*
3. *The program waits for the user to enter data.*
4. *The Scanner reads the data and converts it into the required data type.*

Conceptually, the process can be represented as:

User Input

Scanner Object

Program Processing

Creating a Scanner Object

To use the Scanner class, a Scanner object must be created.
Example:
Scanner sc = new Scanner(System.in);
Here:

- *Scanner is the class name.*

- *sc is the object of the Scanner class.*
- *System.in represents the keyboard input stream.*

Important Methods of Scanner Class

The Scanner class provides several methods for reading different types of input.

nextInt()

Reads an integer value from the input.

Example:

```
int age = sc.nextInt();
```

nextDouble()

Reads a floating-point number from the input.

Example:

```
double price = sc.nextDouble();
```

next()

Reads a single word from the input.

Example:

```
String name = sc.next();
```

nextLine()

Reads an entire line of text including spaces.

Example:

```
String sentence = sc.nextLine();
```

nextFloat()

Reads a float value from the input.

Example of Scanner Class

```
import java.util.Scanner;

public class ScannerExample
{
   public static void main(String[] args)
   {
      Scanner sc = new Scanner(System.in);

      System.out.print("Enter your name: ");
      String name = sc.nextLine();

      System.out.print("Enter your age: ");
      int age = sc.nextInt();

      System.out.println("Name: " + name);
      System.out.println("Age: " + age);
   }
}
```

Explanation

In this example, a Scanner object named sc is created to read input from the keyboard. The program asks the user to enter their name and age.

The nextLine() method reads the name as a string, and the nextInt() method reads the age as an integer. These values are then displayed using output statements.

Characteristics of Scanner Class

The Scanner class has several important characteristics.

- *It can read input from **keyboard, files, or strings**.*
- *It supports multiple data types such as integers, floats, and strings.*
- *It automatically converts input into the required data type.*
- *It uses token-based parsing to process input.*

Advantages of Scanner Class

The Scanner class provides several advantages.

- *It simplifies the process of **reading user input**.*
- *It supports different types of data.*
- *It improves the **interactivity of programs**.*
- *It reduces the need for complex input-handling code.*

Limitations of Scanner Class

Despite its advantages, the Scanner class also has some limitations.

- *It may be slower than some other input methods for very large inputs.*
- *Care must be taken when mixing methods such as nextInt() and nextLine().*

4. Java Input and Output

Introduction

***Input and Output (I/O)** are essential operations in any programming language because programs must interact with users and external systems. **Input** refers to the process of receiving data from the user or other sources, while **Output** refers to the process of displaying or sending processed data to the user or another device.*

In Java, the input and output system is designed to handle different types of data sources such as the keyboard, files, memory, and network connections. Java provides a powerful set of classes that allow programs to read and write data efficiently.

*Most input and output operations in Java are provided through the **java.io**, which contains many classes for handling streams of data. A stream represents a flow of data between a source and a destination.*

For example:

- *Reading input from the keyboard*
- *Writing output to the screen*
- *Reading data from a file*
- *Writing data to a file*

These operations are performed using the classes available in the Java I/O system.

Concept of Java Input and Output

*Conceptually, **Java Input and Output is based on the idea of data streams**. A stream is a sequence of data that flows from one place to another.*

There are two main types of streams:

1. ***Input Stream** – used to read data from a source.*
2. ***Output Stream** – used to send data to a destination.*

For example:

Keyboard → Program → Screen

(Input) (Output)

In this process:

- *The keyboard acts as the **input source**.*
- *The program processes the data.*
- *The screen displays the **output result**.*

Types of Input and Output in Java

Java supports different types of input and output operations depending on the source and destination of the data.

Standard Input

*Standard input refers to data received from the **keyboard**.*

Example:

User enters data → Program reads input

*Classes such as **java.util.Scanner** are commonly used for this purpose.*

4.1 java.io Package

Introduction

*The **java.io** is an important package in Java that provides classes and interfaces for performing **input and output operations**. It allows Java programs to read data from various sources and write data to different destinations.*

In many applications, programs need to handle data from the keyboard, files, memory, or network connections. The java.io package provides a wide range of classes that make it possible to perform these operations efficiently.

*The classes available in this package support both **byte-based and character-based input and output**, allowing programs to handle different types of data such as text, images, and binary files.*

4.2 Java I/O Classes and Interfaces

Introduction

*The **Java Input and Output (I/O) system** provides a large number of classes and interfaces that allow programs to read data from different sources and write data to various destinations. These classes and interfaces are mainly available in the **java.io**.*

In many applications, programs need to interact with external devices such as keyboards, files, printers, or network connections. The Java I/O classes and interfaces provide the necessary mechanisms to perform these operations efficiently.

*The I/O system in Java follows a **stream-based architecture**, where data flows in the form of streams between a source and a destination.*

Concept of Java I/O Classes and Interfaces

*Conceptually, the Java I/O system is organized using **interfaces and classes that manage the flow of data**.*

- ***Interfaces** define the structure and basic behavior of input and output operations.*
- ***Classes** provide the actual implementation of these operations.*

The system is designed in such a way that base classes provide general functionality, and specialized classes extend these base classes to perform specific tasks such as file handling, buffering, or data conversion.

*Thus, the Java I/O system forms a **hierarchical framework of classes and interfaces** that work together to handle different types of input and output operations.*

Types of Java I/O Streams

Java organizes its I/O system into two main categories of streams:

1. ***Byte Streams***
2. ***Character Streams***

These streams help Java handle both binary data and text data efficiently.

4.4.1 Byte Streams

Introduction

*In Java, **Byte Streams** are used to perform input and output operations involving **binary data**. Binary data refers to data that is stored in the form of bytes rather than characters. Examples include image files, audio files, video files, and other types of non-text data.*

*Byte streams process data in the form of **8-bit bytes** and transfer this data between a program and an external source or destination. These streams are useful when a program needs to read or write raw data without interpreting it as characters.*

*The classes related to byte streams are available in the **java.io**. The two main base classes that support byte stream operations are:*

- ***java.io.InputStream***
- ***java.io.OutputStream***

These classes provide the basic functionality required for reading and writing byte data.

Byte Streams

*byte stream can be understood as **a continuous flow of bytes between a data source and a data destination**. When a program reads data from a file or device using byte streams, the data is transferred **one byte at a time**.*

For example, when an image file is read by a program, the file's binary data is transferred as a sequence of bytes through an input stream.

The process can be illustrated as follows:

Data Source → InputStream → Java Program → OutputStream → Data Destination

In this flow:

- *The **InputStream** reads byte data from a source.*
- *The program processes the data.*
- *The **OutputStream** sends the processed data to a destination.*

Thus, byte streams serve as a communication channel that transfers binary data between different components of a system.

Characteristics of Byte Streams

Byte streams have several important characteristics.

Binary Data Handling

Byte streams are designed to handle raw binary data rather than text.

Byte-Oriented Processing

Data is processed in units of bytes, which makes these streams suitable for handling files such as images and multimedia data.

Sequential Data Transfer

Data flows sequentially from the source to the destination.

Platform Independence

Java byte streams provide a consistent way of handling binary data across different operating systems.

Base Classes of Byte Streams

The Java byte stream system is built on two fundamental base classes.

InputStream

*The **InputStream** class is used to read byte data from an input source such as a file, network connection, or device.*

Several classes extend InputStream to provide specific input functionalities. Examples include FileInputStream and BufferedInputStream.

OutputStream

*The **OutputStream** class is used to write byte data to an output destination such as a file or network connection.*

Subclasses such as FileOutputStream and BufferedOutputStream extend this class to provide specialized output capabilities.

Common Byte Stream Classes

Some commonly used classes related to byte streams include:

- ***FileInputStream** – used to read bytes from a file.*
- ***FileOutputStream** – used to write bytes to a file.*
- ***BufferedInputStream** – reads data efficiently by using a buffer.*
- ***BufferedOutputStream** – writes data efficiently using buffered output.*

These classes improve the efficiency and performance of input and output operations.

Example of Byte Stream

```
import java.io.FileInputStream;

public class ByteStreamExample
{
   public static void main(String[] args) throws Exception
   {
      FileInputStream fin = new FileInputStream("data.txt");

      int data;

      while((data = fin.read()) != -1)
      {
         System.out.print((char)data);
      }

      fin.close();
   }
}
```

Explanation

*In this example, a **FileInputStream** object is created to read data from a file named data.txt. The read() method retrieves data from the file one byte at a time. The loop continues until the end of the file is reached. Each byte is then converted into a character and displayed on the screen.*

Advantages of Byte Streams

Byte streams provide several advantages when working with binary data.

- *They allow programs to handle **binary files efficiently**.*
- *They support reading and writing **raw data directly**.*
- *They can be used with various input and output sources such as files and networks.*

Limitations of Byte Streams

Despite their usefulness, byte streams also have some limitations.

- *They are not well suited for handling text data that requires character encoding.*
- *For processing text data, **character streams** are usually more appropriate.*

Character Stream Classes

4.4.2 Character Stream Classes

Introduction

***Character Stream Classes** in Java are used to perform input and output operations involving **text data**. These streams handle data in the form of **characters rather than raw bytes**, which makes them suitable for reading and writing text files.*

*Character streams are particularly useful when dealing with **Unicode characters**, because Java internally uses Unicode to represent text. This allows programs to work with characters from different languages and character sets.*
*The classes related to character streams are available in the **java.io**. The two fundamental base classes for character stream operations are:*

- ***java.io.Reader***
- ***java.io.Writer***

These classes provide methods for reading and writing characters from various input and output sources.

Concept of Character Streams
*Conceptually, character streams represent **a flow of text data between a source and a destination**. Instead of transferring data byte by byte, character streams process data **character by character**.*
For example, when a program reads a text file, each character in the file is read sequentially and processed by the program.
The general flow of character stream operations can be represented as follows:
Text File → Reader → Java Program → Writer → Output Destination
In this process:

- *The **Reader** class reads characters from a source.*
- *The program processes the characters.*
- *The **Writer** class sends characters to an output destination.*

Characteristics of Character Streams
Character streams have several important characteristics.
Character-Oriented Processing
These streams process data in the form of characters instead of raw bytes.
Unicode Support
Character streams support Unicode encoding, which allows programs to handle text from multiple languages.
Suitable for Text Files
They are mainly used for reading and writing text files.
Sequential Data Transfer
Characters are transferred sequentially from the input source to the output destination.

Base Classes of Character Streams
The character stream system in Java is built on two primary base classes.
Reader
*The **Reader** class is used to read characters from input sources such as text files or other character-based streams.*
Several classes extend the Reader class to provide specialized functionality. Examples include FileReader and BufferedReader.

Writer

*The **Writer** class is used to write characters to output destinations such as text files or other character-based streams.*
Subclasses such as FileWriter and BufferedWriter extend this class to provide additional capabilities.

Common Character Stream Classes

Some commonly used character stream classes include:

- ***FileReader** – reads characters from a file.*
- ***FileWriter** – writes characters to a file.*
- ***BufferedReader** – reads text efficiently using a buffer.*
- ***BufferedWriter** – writes text efficiently using buffering.*

These classes help improve the efficiency and convenience of text-based input and output operations.

Example of Character Stream

```
import java.io.FileReader;

public class CharacterStreamExample
{
   public static void main(String[] args) throws Exception
   {
      FileReader fr = new FileReader("data.txt");

      int ch;

      while((ch = fr.read()) != -1)
      {
         System.out.print((char)ch);
      }

      fr.close();
   }
}
```

Explanation

*In this example, a **FileReader** object is created to read characters from a text file named data.txt. The read() method reads one character at a time. The loop continues until the end of the file is reached.*
Each character is then displayed on the screen.

Advantages of Character Streams

Character streams provide several advantages for handling text data.

- *They support **Unicode characters**, allowing programs to handle multiple languages.*
- *They are well suited for **reading and writing text files**.*
- *They simplify the processing of character-based data.*

Limitations of Character Streams

Despite their advantages, character streams also have some limitations.

- *They are not suitable for handling **binary data such as images or audio files**.*
- *Byte streams are more appropriate for binary file operations.*

4.3 File Class

Introduction

*The **java.io.File** is an important class in Java used for **working with files and directories**. It is part of the **java.io** and provides methods that allow Java programs to create, delete, rename, and obtain information about files and directories.*

*In many applications, data must be stored permanently in files. The File class helps Java programs interact with the file system of the computer. It does not directly read or write data from files; instead, it provides methods for **managing file and directory paths**.*

For actual reading and writing operations, other classes such as FileInputStream or FileOutputStream are used along with the File class.

File Class

*Conceptually, the File class represents **an abstract representation of a file or directory path in the file system**. It acts as a bridge between a Java program and the physical file system of the computer.*

*When a File object is created, it does not necessarily create a physical file immediately. Instead, it simply represents the **location of a file or directory**.*

For example, a File object may represent the following path:

C:\Documents\example.txt

In this case, the File object stores the path information and allows the program to perform operations such as checking whether the file exists or creating a new file.

Characteristics of File Class

The File class has several important characteristics.

Represents File or Directory Path

A File object represents a path that may correspond to either a file or a directory.

Does Not Directly Handle File Content

The File class does not read or write file data. It only provides methods to manipulate file properties.

Platform Independent

Java handles file paths in a platform-independent manner so that programs can run on different operating systems.

Supports File Management Operations

The File class provides methods to create, delete, rename, and check properties of files.

Creating a File Object

A File object can be created using its constructor.

Example:

File file = new File("example.txt");

*In this example, the object file represents a file named **example.txt**. However, the file is not created physically until specific methods are used.*

Common File Operations in Java

Several important file operations are commonly performed in Java programs.

1. Creating a File

Creating a file means generating a new file in the file system. The File class provides the createNewFile() method for this purpose.

Example:

```
import java.io.File;
import java.io.IOException;

public class CreateFileExample
{
   public static void main(String[] args) throws IOException
   {
      File file = new File("example.txt");

      if(file.createNewFile())
      {
         System.out.println("File created successfully.");
      }
      else
      {
         System.out.println("File already exists.");
      }
   }
}
```

---

2. Writing Data to a File

Writing to a file means storing information inside a file. This operation is commonly performed using classes such as FileWriter.

Example:

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteFileExample
{
   public static void main(String[] args) throws IOException
   {
```

```
        FileWriter writer = new FileWriter("example.txt");

        writer.write("Java File Handling Example");

        writer.close();
    }
}
```

---

3. Reading Data from a File

Reading a file means retrieving information stored in the file. Classes such as FileReader and BufferedReader are used for this purpose.

Example:

```
import java.io.FileReader;
import java.io.IOException;

public class ReadFileExample
{
    public static void main(String[] args) throws IOException
    {
        FileReader reader = new FileReader("example.txt");

        int ch;
        while((ch = reader.read()) != -1)
        {
            System.out.print((char)ch);
        }

        reader.close();
    }
}
```

---

4. Renaming a File

A file can be renamed using the renameTo() method of the File class.

Example:

```
File oldFile = new File("example.txt");
File newFile = new File("newfile.txt");

oldFile.renameTo(newFile);
```

---

5. Deleting a File

Files that are no longer required can be removed from the system using the delete() method.

Example:

```
File file = new File("example.txt");

if(file.delete())
{
   System.out.println("File deleted successfully.");
}
else
{
   System.out.println("Unable to delete the file.");
}
```

Advantages of File Class

The File class offers several advantages.

- *Provides methods for **file and directory management***
- *Helps programs interact with the file system*
- *Supports operations such as **creating, deleting, and renaming files***
- *Works across different operating systems*

Limitations of File Class

Despite its usefulness, the File class has certain limitations.

- *It does not support direct reading or writing of file content*
- *Additional classes are required to perform data input and output operations*
- *Some commonly used methods for file operations are explained below.*

Method	Description
createNewFile()	*This method is used to create a new file in the file system. If the file already exists, the method returns false.*
exists()	*This method checks whether a file or directory exists at the specified path. It returns true if the file exists and false otherwise.*
delete()	*This method deletes the specified file from the file system. If the deletion is successful, it returns true.*
renameTo()	*This method is used to rename an existing file or move it to another location.*
length()	*This method returns the size of the file in bytes. It helps determine how much data is stored in the file.*
canRead()	*This method checks whether the file has permission to be read by the program.*
canWrite()	*This method checks whether the file has permission to be modified or written.*

4.5 Serialization

Introduction

Serialization *is an important mechanism in Java that allows an object to be* ***converted into a sequence of bytes*** *so that it can be stored in a file or transmitted over a network. In object-*

*oriented programming, objects contain both **data (fields)** and **behavior (methods)**.*
*Serialization focuses on preserving the **state of the object**, which means the values stored in its variables.*
*When an object is serialized, its state is converted into a **byte stream representation** that can be written to a storage medium such as a file or sent through a network connection. Later, this byte stream can be converted back into the original object. This makes serialization extremely useful in applications that require **data persistence, distributed communication, and object storage**.*
*Serialization is supported by classes available in the **java.io**. For an object to be serialized, its class must implement the **java.io.Serializable** interface.*

Concept of Serialization
*Conceptually, serialization can be understood as a process that **transforms an object into a stream of bytes so that it can be stored or transferred**. The byte stream contains the information required to recreate the object later.*
When serialization occurs, the Java Virtual Machine (JVM) converts the object's internal data into a format that can be saved or transmitted. The object structure and variable values are encoded into the byte stream.
The general concept can be represented as:
Object → Serialization → Byte Stream → Storage / Transmission
*Later, the stored byte stream can be converted back into the original object using a reverse process known as **deserialization**.*
*Thus, serialization enables programs to **preserve the state of objects beyond the lifetime of the program execution**.*

Internal Working of Serialization
*Internally, serialization works by capturing the **state of an object and its associated data members** and converting them into a byte stream.*
During serialization, the following steps occur:

1. *The object is passed to a serialization stream.*
2. *The Java runtime examines the object and verifies that the class implements the Serializable interface.*
3. *The values of all non-transient instance variables are converted into a byte stream.*
4. *The byte stream is written to a file or network output stream.*

This process allows complex objects to be stored or transmitted without manually writing code for each data member.

Purpose of Serialization
Serialization serves several important purposes in Java applications.
Object Persistence
Serialization allows objects to be stored permanently in files so that they can be retrieved later.
Data Transmission

Objects can be transmitted over networks in distributed applications.

Saving Object State

The current state of an object can be preserved and restored later.

Communication Between Systems

Serialization allows objects to be transferred between different programs or systems.

Remote Communication

Serialization is widely used in technologies such as remote method invocation and distributed computing.

Serializable Interface

*To enable serialization, a class must implement the **Serializable interface**. This interface does not contain any methods; instead, it acts as a **marker interface**.*

A marker interface is an interface that signals the Java runtime that a particular class supports a specific capability. In this case, the Serializable interface informs the Java Virtual Machine that objects of the class can be converted into a byte stream.

Example:

```
import java.io.Serializable;

class Student implements Serializable
{
   int id;
   String name;
}
```

In this example, the Student class implements the Serializable interface, allowing its objects to participate in serialization.

ObjectOutputStream Class

*The **ObjectOutputStream** class is used to write serialized objects to an output stream. It converts objects into a byte stream and writes them to a file or another output destination.*

Example:

```
import java.io.*;

class SerializationExample
{
   public static void main(String args[]) throws Exception
   {
      Student s = new Student();
      s.id = 101;
      s.name = "Ravi";

      FileOutputStream fout = new FileOutputStream("student.dat");
      ObjectOutputStream out = new ObjectOutputStream(fout);
```

```
        out.writeObject(s);

        out.close();
        fout.close();

        System.out.println("Object Serialized Successfully");
    }
}
```

In this program, the object s is converted into a byte stream and stored in the file student.dat.

Deserialization

Deserialization *is the reverse process of serialization. It converts the stored byte stream back into the original object.*

This process is performed using the ***ObjectInputStream*** *class. During deserialization, the JVM reconstructs the object by reading the stored byte stream and restoring the object's state.*

Example:

```
import java.io.*;

class DeserializationExample
{
    public static void main(String args[]) throws Exception
    {
        FileInputStream fin = new FileInputStream("student.dat");
        ObjectInputStream in = new ObjectInputStream(fin);

        Student s = (Student) in.readObject();

        System.out.println(s.id + " " + s.name);

        in.close();
        fin.close();
    }
}
```

Here, the serialized object stored in the file is read and converted back into the original Student object.

Advantages of Serialization

Serialization provides several benefits in Java programming.

- *It allows objects to be* ***stored permanently in files****.*
- *It enables* ***efficient transmission of objects over networks****.*
- *It simplifies the process of saving and restoring complex objects.*
- *It supports* ***distributed computing and remote communication****.*

Limitations of Serialization

Despite its usefulness, serialization also has certain limitations.

- *Serialized data is stored in **binary format**, making it difficult for humans to read.*
- *Changes in class structure may cause compatibility issues with previously serialized objects.*
- *Serialization may introduce **performance overhead** when large objects are processed.*

Smart Study PLUS

1. **Swing**
   - 1.1 JFrame
   - 1.2 JLabel
   - 1.3 ImageIcon
   - 1.4 JTextField
   - 1.5 JPasswordField
   - 1.6 Swing Buttons
   - 1.7 JTabbedPane
   - 1.8 JScrollPane
   - 1.9 JList
   - 1.10 JComboBox
2. **Event Handling**
   - 2.1 Event Classes
   - 2.2 Source of Events
   - 2.3 Event Listener Interfaces
   - 2.4 Mouse Events
   - 2.5 Keyboard Events
3. **JavaFX**
   - 3.1 Introduction to JavaFX
   - 3.2 JavaFX Setup
   - 3.3 Scene Graph
     - 3.3.1 Root
     - 3.3.2 Children
     - 3.3.3 Hierarchy
   - 3.4 Layout Containers
     - 3.4.1 VBox
     - 3.4.2 HBox
     - 3.4.3 GridPane
     - 3.4.4 StackPane
   - 3.5 Buttons and Controls
   - 3.6 Styling in JavaFX
   - 3.7 Animation and Transitions in JavaFX

Swing (javax.swing)

```
|
├── Top Level Containers
|   └── JFrame
|
├── Basic Components
|   ├── JLabel
```

```
│  ├── ImageIcon
│  ├── JTextField
│  └── JPasswordField
│
├── Buttons
│  └── JButton
│
├── Containers
│  ├── JTabbedPane
│  └── JScrollPane
│
└── Selection Components
├── JList
└── JComboBox
```

1.1 Swing (javax.swing)

Introduction

Swing *is a graphical user interface (GUI) toolkit in Java that is used to develop* ***window-based desktop applications****. It provides a collection of components such as windows, buttons, labels, text fields, lists, and menus that allow users to interact with programs through a graphical interface.*

Swing is part of the ***javax.swing*** *package. It is built on top of the* ***java.awt*** *library but offers more advanced and flexible components.*

Concept of Swing

Swing *can be viewed as a framework that provides a collection of* ***predefined graphical components used to build user interfaces in Java applications****. These components help developers design programs that allow users to interact with the application through visual elements rather than text commands.*

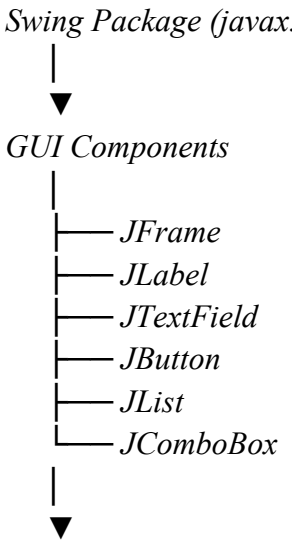
In a Swing-based program, different components such as windows, buttons, labels, and text fields are arranged inside containers to create a structured interface. These components work together to form a complete graphical environment where users can enter information, view results, and perform various operations.

Thus, Swing acts as a ***toolkit for designing interactive desktop applications****, enabling developers to combine multiple graphical elements to create organized and user-friendly interfaces.*

In a Swing application, the interface is built by placing various components inside containers such as frames and panels. These components allow users to enter data, view information, and interact with the program.

The general structure of a Swing application can be represented as:

Java Program



Features of Swing

- *Swing components are platform independent because they are written completely in Java and can run on different operating systems.*
- *Swing provides lightweight components that do not depend on the native operating system components.*
- *It offers a rich set of graphical components such as JFrame, JButton, JLabel, JTextField, JList, and JComboBox for building user interfaces.*
- *Swing supports pluggable look and feel, which allows the appearance of the application to be changed easily.*
- *Swing components are highly customizable, allowing developers to modify their appearance and behavior according to program requirements.*
- *Swing follows the Model–View–Controller (MVC) architecture, which separates data, user interface, and control logic.*
- *It supports event-driven programming, enabling programs to respond to user actions such as mouse clicks and keyboard input.*
- *Swing provides an extensible framework that allows programmers to create new components by extending existing classes.*

Common Swing Components

Some commonly used Swing components include:

- ***JFrame** – main window of a GUI application*
- ***JLabel** – displays text or images*
- ***JTextField** – allows user text input*
- ***JPasswordField** – allows secure password input*
- ***JButton** – clickable button used to perform actions*
- ***JList** – displays a list of selectable items*
- ***JComboBox** – displays a drop-down list of options*

1.1 JFrame

Introduction

JFrame *is one of the most fundamental classes in the Swing library used to create the* ***main window of a graphical user interface (GUI) application****. It provides the basic structure for building window-based applications in Java. Every Swing application generally begins with a JFrame because it acts as the* ***primary container that holds and displays all other GUI components****.*

The JFrame class belongs to the ***javax.swing****, which provides various graphical components for creating desktop applications. It is built on top of the* ***java.awt*** *framework and inherits many features from the AWT library.*

A JFrame represents a ***top-level container****, meaning it is not placed inside another container. Instead, it serves as the outermost window that appears on the screen and allows the user to interact with the application.*

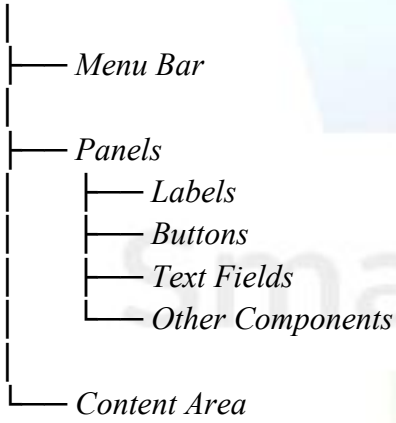
Concept of JFrame

Conceptually, a ***JFrame can be understood as the main window that forms the foundation of a graphical interface****. It acts as a container where different components such as labels, buttons, text fields, menus, and panels are placed.*

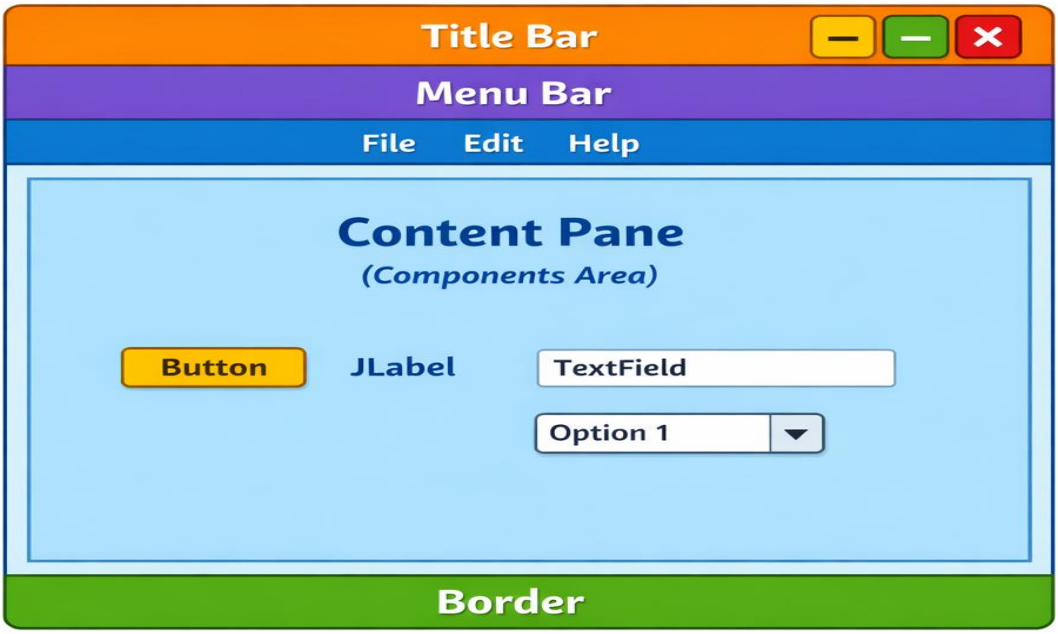
In a Swing application, all graphical components are organized inside containers, and the JFrame is usually the ***highest level container*** *that manages the layout and display of these components.*

The conceptual structure of a JFrame can be illustrated as follows:

JFrame (Main Application Window)



Thus, the JFrame acts as ***the outer frame that organizes and displays all graphical elements of the application****.*



Internal Structure of JFrame

Internally, a JFrame contains several important parts that work together to display the graphical interface.

Title Bar

The title bar appears at the top of the window and displays the name of the application. It also contains control buttons such as minimize, maximize, and close.

Content Pane

The content pane is the main area where components such as buttons, labels, and text fields are added.

Menu Bar

A menu bar can be attached to the frame to provide menu options for the user.

Border

The border surrounds the frame and defines the boundaries of the window.

These elements together form the complete graphical window that the user interacts with.

Characteristics of JFrame

JFrame has several important characteristics that make it essential for GUI development.

Top-Level Container

JFrame acts as a top-level container that holds and manages other components.

Supports Multiple Components

Various GUI components such as labels, buttons, panels, and text fields can be placed inside a frame.

Customizable Window

Developers can set properties such as window size, title, layout, and visibility.

Window Control Options

A JFrame provides standard window controls such as minimize, maximize, and close operations.

Supports Layout Management

Different layout managers can be used to arrange components inside the frame.

Important Methods of JFrame

The JFrame class provides several methods for managing the window and its components.

setTitle(String title)

Sets the title displayed on the title bar of the window.

setSize(int width, int height)

Defines the width and height of the frame.

setVisible(boolean value)

Controls whether the frame is visible on the screen.

add(Component c)

Adds a graphical component to the frame.

setLayout(LayoutManager manager)

Specifies how components should be arranged inside the frame.

setDefaultCloseOperation(int operation)

Defines what happens when the user closes the window.

Example of JFrame

```
import javax.swing.*;

public class JFrameExample
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Swing JFrame Example");

      frame.setSize(400,300);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Explanation

*In this program, a **JFrame object** named frame is created to represent the application window. The setTitle() method defines the title displayed in the title bar. The setSize() method specifies the width and height of the window.*

The setDefaultCloseOperation() method ensures that the application stops running when the user closes the window. Finally, setVisible(true) makes the frame visible to the user.

Advantages of Using JFrame

Using JFrame in Java GUI applications provides several advantages.

- *It provides the **main window structure** for graphical applications.*
- *It allows the integration of multiple GUI components within a single window.*
- *It supports flexible layout management for arranging components.*
- *It enables developers to create interactive and user-friendly applications.*

Basic Components

*In the Swing framework, **basic components** are fundamental graphical elements that allow a program to display information and receive input from the user. These components are essential for building graphical user interfaces because they represent the interactive elements that users see and interact with on the screen.*

*Basic components are usually placed inside containers such as **JFrame or JPanel**, where they are arranged according to a layout manager. Each component performs a specific role, such as displaying text, showing images, or accepting user input. By combining multiple components together, developers can create structured and interactive user interfaces.*

*Some of the most commonly used basic components in Swing include **JLabel, ImageIcon, JTextField, and JPasswordField**. Each of these components serves a particular purpose within the interface.*

JLabel

***JLabel** is a Swing component used to display **non-editable text, images, or a combination of both** within a graphical interface. It is commonly used to present information, instructions, or descriptions to the user.*

*Labels do not accept input from the user; instead, they serve as **display elements** that help explain the purpose of other components. For example, a label may be used to identify a text field where the user enters information.*

JLabel can also display icons or images, which improves the visual appearance of an application. It supports features such as text alignment, font customization, and image positioning.

Example concept:

Username: [_____]
Password: [_____]

In this example, ***"Username" and "Password" are displayed using JLabel components****.*

ImageIcon

ImageIcon *is a class used to* ***load and display images in Swing components****. It allows applications to include graphical elements such as icons, logos, or pictures within the user interface.*

ImageIcon objects are usually used together with components such as ***JLabel, JButton, or JMenuItem****. When an image file is loaded using ImageIcon, it can be assigned to a component to display that image on the screen.*

For example, an application may display a company logo at the top of a window or use icons on buttons to represent different actions.

Conceptually, the process works as follows:

Image File → ImageIcon Object → Displayed in JLabel / JButton

This helps make the graphical interface more attractive and easier for users to understand.

JTextField

JTextField *is a Swing component used to* ***accept a single line of text input from the user****. It provides a small editable field where users can type characters using the keyboard.*

This component is commonly used in forms where users must enter information such as names, email addresses, or numeric values. JTextField allows the program to retrieve the entered data and process it as needed.

For example, a login form may contain a text field for entering a username.

Example concept:

Name: [_____]

The blank area represents the ***JTextField where the user enters information****.*

JTextField also supports operations such as setting default text, controlling the number of characters displayed, and retrieving the entered text.

JPasswordField

***JPasswordField** is a special type of text field designed specifically for **entering passwords securely**. It functions similarly to JTextField but hides the characters typed by the user.*

*Instead of displaying the actual characters, the component replaces them with symbols such as * or •. This ensures that sensitive information remains protected while being entered.*

Example concept:

*Password: ********

Although the characters are hidden on the screen, the program can still retrieve the entered password when required.

JPasswordField is widely used in login systems, authentication forms, and applications where secure data entry is necessary.

Buttons

*In the Swing framework, **buttons** are interactive components that allow users to perform specific actions in a graphical user interface. A button acts as a control element that triggers an event when the user clicks it. Buttons are commonly used in applications for tasks such as submitting information, opening files, starting operations, or canceling actions.*

*In Swing, buttons are mainly represented by the **JButton** class. This class provides a graphical button that users can click using a mouse or activate using the keyboard. When the button is pressed, it generates an event that the program can detect and respond to through event-handling mechanisms.*

*Buttons are usually placed inside containers such as **JFrame or JPanel**, and they often display text, icons, or both. The text on the button usually indicates the action that will be performed when the button is clicked.*

For example, common buttons in applications include:

[Login] [Submit] [Cancel] [Save]

Each of these buttons performs a different function depending on the design of the program.

*The **JButton** component also allows developers to customize its appearance and behavior. Developers can set properties such as the button label, size, position, font style, and color. Additionally, icons can be added to buttons to make them visually appealing and easier to understand.*

*Buttons play a crucial role in graphical user interfaces because they provide **a simple and intuitive way for users to interact with applications**. By clicking buttons, users can execute commands and control the flow of the program.*

JButton Program (Example)

```
import javax.swing.*;

public class JButtonExample
```

```
{
   public static void main(String[] args)
   {
      // Create a frame
      JFrame frame = new JFrame("JButton Example");

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

      // Set position and size of the button
      button.setBounds(120,100,120,40);

      // Add button to the frame
      frame.add(button);

      // Set frame size
      frame.setSize(400,300);

      // Use no layout manager
      frame.setLayout(null);

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

Explanation

*In this program, a **JFrame** is created to represent the main window of the application. A **JButton** component is then created with the label **"Click Me"**. The setBounds() method is used to define the position and size of the button inside the frame.*

The button is added to the frame using the add() method. Finally, the frame size is set and the window is made visible using the setVisible(true) method. When the program runs, a window appears on the screen containing the button.

Containers

*In the Swing framework, **containers** are special components that are used to **hold and organize other graphical components** inside a graphical user interface. Containers help manage the layout and arrangement of different components such as buttons, labels, and text fields.*

Containers play an important role in GUI design because they allow developers to group related components together and control how these components appear within the application window. By using containers, complex user interfaces can be structured in an organized and systematic way.

*Some commonly used container components in Swing include **JTabbedPane** and **JScrollPane**.*

JTabbedPane

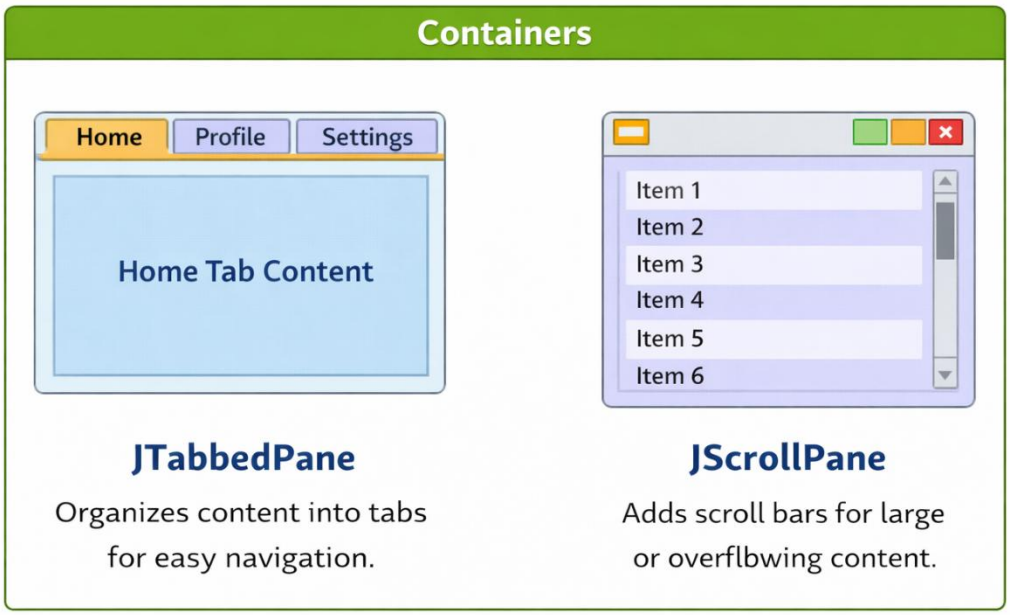
***JTabbedPane** is a container component that allows multiple panels or components to be organized into **separate tabs within a single window**. Each tab represents a different section of the interface, and users can switch between these sections by clicking the tabs.*

This component is useful when an application needs to display several groups of information without opening multiple windows. Instead, each group is placed in a different tab.

For example, an application may contain tabs such as:

[Home] [Profile] [Settings]

When the user selects a tab, the content related to that tab becomes visible. This helps organize information in a clear and efficient way.



JScrollPane

JScrollPane *is a container component that provides* ***scroll bars for other components when their content exceeds the visible area****. It allows users to navigate through large amounts of content by scrolling vertically or horizontally.*

This component is commonly used with elements such as text areas, lists, tables, or images that may contain more information than can be displayed in a fixed window size.

For example, if a list contains many items, a scroll pane allows the user to move through the list using scroll bars.

Conceptually, the structure can be represented as:

JScrollPane

└── Component with Large Content

Thus, JScrollPane ensures that large content can be viewed conveniently without increasing the size of the application window.

Selection Components

In the Swing framework, ***selection components*** *are graphical elements that allow users to* ***choose one or more options from a set of available items****. These components are commonly used in graphical interfaces where the user needs to select data from a list instead of typing it manually.*

Selection components improve usability because they provide predefined choices and reduce the possibility of incorrect input. Two commonly used selection components in Swing are ***JList*** *and* ***JComboBox****.*

JList

JList *is a Swing component used to display* ***a list of items from which the user can select one or multiple options****. The items in the list are usually displayed vertically.*

JList is useful when an application needs to present several options to the user. The user can select an item by clicking on it with the mouse. Depending on the configuration, the component may allow either ***single selection or multiple selections****.*

Example concept:

Programming Languages

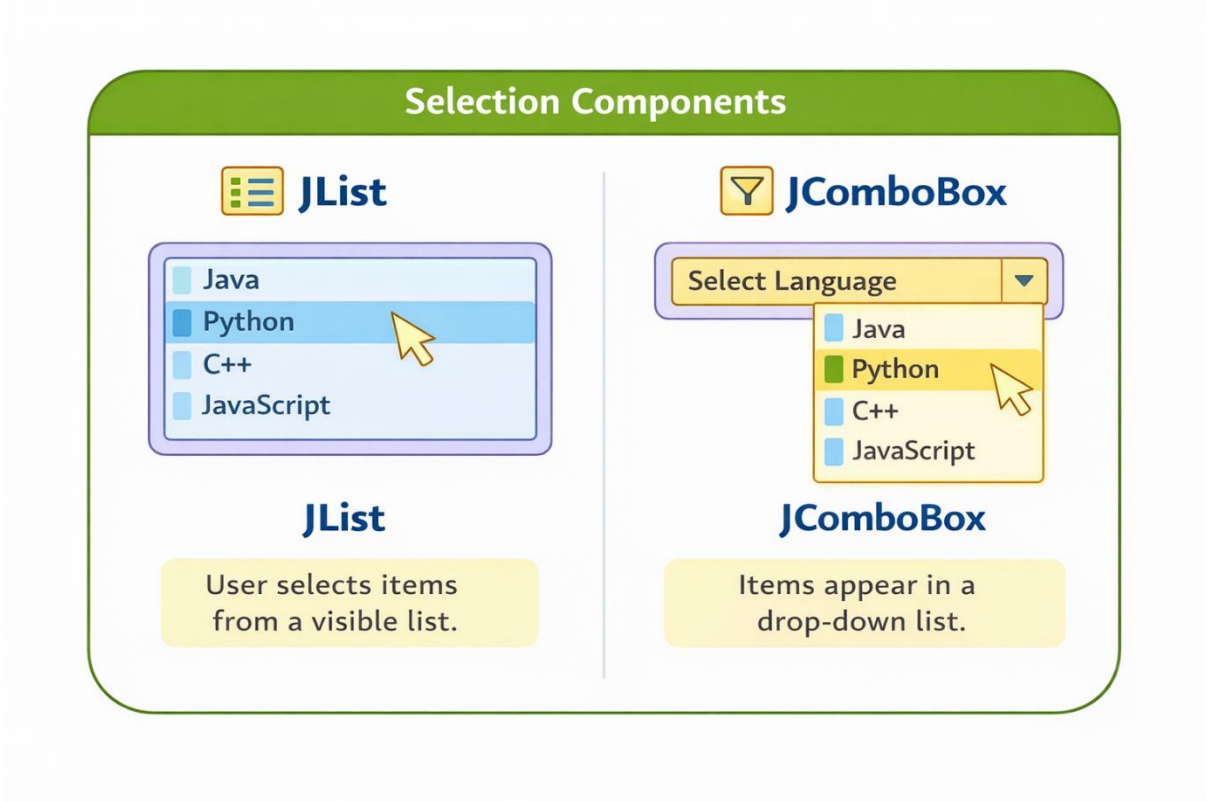
• Java

• Python

• C++

• JavaScript

In this example, the user can select one or more programming languages from the list.



JComboBox

JComboBox *is a Swing component that provides a* ***drop-down list of items*** *from which the user can select one option. It combines the features of a text field and a list.*

When the user clicks the arrow on the combo box, a list of options appears. The user can then choose the desired option from the list.

Example concept:

Select Country

[India ▼]

When the drop-down arrow is clicked, a list of countries appears for selection.

2. Event Handling

Introduction

*In graphical user interface (GUI) programming, applications must respond to actions performed by the user. These actions may include clicking a button, typing on the keyboard, selecting items from a list, moving the mouse, or closing a window. Each of these actions produces an **event**. The mechanism used by a program to detect these events and respond to them appropriately is known as **event handling**.*

*Event handling plays a crucial role in GUI applications because it allows programs to become **interactive and responsive**. Without event handling, graphical components such as buttons or menus would not perform any operations when the user interacts with them.*

*In Java, the event handling mechanism is implemented through a set of classes and interfaces provided in the **java.awt.event**. These classes define various types of events and the methods used to handle them.*

Concept of Event Handling

*Conceptually, event handling is the process through which a program **detects user-generated events and executes appropriate code in response to those events**. When a user performs an action on a graphical component, the system generates an event object that contains information about that action.*

*The event is then passed to a special object called a **listener**, which contains the logic for handling that event.*

The overall interaction can be represented conceptually as:

User Interaction

Event Source (GUI Component)

Event Object Created

Event Listener Receives Event

Event Handling Code Executes

Program Performs Required Action

*Thus, event handling acts as a **communication mechanism between the user and the program**.*

Event Delegation Model

*Java uses an **Event Delegation Model** to manage events. In this model, the responsibility of handling an event is delegated to a separate object called a listener.*
Instead of the component itself processing the event, the component simply generates the event and forwards it to the registered listener object. This design makes the program more organized and modular.
The event delegation model consists of three important elements:

1. ***Event Source***
2. ***Event Object***
3. ***Event Listener***

Event Source
*An **event source** is the object that generates the event. It is usually a graphical component with which the user interacts.*
Examples of event sources include:

- *JButton*
- *JTextField*
- *JMenuItem*
- *JList*
- *JComboBox*

When a user interacts with these components, they generate events such as button clicks or text input.
For example, when a user presses a button, the button acts as the event source and generates an event.

Event Object
*An **event object** contains detailed information about the event that has occurred. It describes the type of event and the source that generated it.*
Different types of events in Java are represented by different classes.
Some common event classes include:

- ***ActionEvent** – generated when a button is clicked.*
- ***MouseEvent** – generated when mouse actions occur.*
- ***KeyEvent** – generated when keyboard keys are pressed or released.*
- ***WindowEvent** – generated when a window is opened or closed.*

The event object is automatically created by the system when an event occurs.

Event Listener
*An **event listener** is an interface that contains methods used to respond to specific types of events. A listener waits for an event to occur and performs a specific action when the event is detected.*
*For example, when a button is clicked, an **ActionListener** can be used to respond to the event.*
Each listener interface contains methods that must be implemented by the program. These methods define how the program should react to the event.

Example:
ActionListener → actionPerformed()
MouseListener → mouseClicked(), mousePressed()
KeyListener → keyPressed(), keyReleased()

Event Registration

*For event handling to work, the listener must be **registered with the event source**. This process is known as **event registration**.*
During registration, the program informs the event source which listener should receive notifications when an event occurs.
For example:
button.addActionListener(listenerObject);
This statement registers the listener with the button so that it receives events when the button is clicked.

Event Handling Process

The complete process of event handling in Java generally follows these steps:

1. *A GUI component such as a button is created.*
2. *The component acts as the **event source**.*
3. *A listener object is created to handle the event.*
4. *The listener is registered with the event source.*
5. *When the user interacts with the component, an event object is created.*
6. *The listener receives the event and executes the corresponding method.*

This mechanism ensures that the program reacts correctly to user interactions.

Example of Event Handling

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

public class EventHandlingExample
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Event Handling Example");

      JButton button = new JButton("Click Me");
      button.setBounds(120,100,120,40);

      button.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent e)
         {
            System.out.println("Button Clicked");
```

```
        }
    });

    frame.add(button);
    frame.setSize(400,300);
    frame.setLayout(null);
    frame.setVisible(true);
  }
}
```

Explanation

*In this program, a **JButton** is created inside a JFrame window. The button acts as the **event source**. An **ActionListener** is attached to the button using the addActionListener() method. When the user clicks the button, an **ActionEvent** is generated. The listener detects this event and executes the actionPerformed() method, which prints the message **"Button Clicked"**.*

Advantages of Event Handling

Event handling provides several advantages in GUI applications.

- *It enables **interactive user interfaces**.*
- *It allows programs to respond to user actions dynamically.*
- *It separates the user interface logic from event processing logic.*
- *It improves program structure and maintainability.*

2.1 Event Classes

Introduction

*In Java event handling, **event classes** represent different types of events that occur when a user interacts with graphical components of a program. An event is an action that takes place during the execution of a program, such as clicking a button, pressing a key, moving the mouse, or opening a window.*

*Whenever such an action occurs, Java automatically creates an **event object** that contains information about the event. The class used to create this event object is known as an **event class**. Event classes are mainly defined in the **java.awt.event**.*

These classes store details about the event, including the source of the event, the type of action performed, and other related information.

Concept of Event Classes

*Conceptually, an event class can be understood as **a structure that represents a specific type of user action in a program**. When a user performs an action, Java creates an object of the corresponding event class and passes it to the appropriate event listener.*

This process allows the program to identify the type of event and respond accordingly.

The conceptual flow can be represented as:

User Action

Event Occurs

Event Object Created (Event Class)

Event Listener Receives Event

Program Executes Response

Thus, event classes serve as ***information carriers that describe what action occurred and where it occurred****.*

Common Event Classes

Java provides several event classes that represent different types of user interactions. Some commonly used event classes include:

ActionEvent

ActionEvent occurs when a user performs an action such as clicking a button, selecting a menu item, or pressing the Enter key in a text field.

MouseEvent

MouseEvent occurs when the user interacts with the mouse. Examples include clicking the mouse button, moving the mouse pointer, or dragging the mouse.

KeyEvent

KeyEvent occurs when a user presses, releases, or types a key on the keyboard.

WindowEvent

WindowEvent occurs when a window operation takes place, such as opening, closing, minimizing, or maximizing a window.

ItemEvent

ItemEvent occurs when the state of an item changes, such as selecting or deselecting a checkbox or selecting an option in a combo box.

Role of Event Classes in Event Handling

Event classes play a significant role in the event handling mechanism. They help the program identify the type of event that has occurred and provide the necessary information required for handling that event.

Each event object contains details such as:

- *the source component that generated the event*
- *the time at which the event occurred*
- *the type of user action performed*

This information allows the program to execute the correct response.

2.1 Event Classes

Introduction

In Java graphical user interface programming, ***event classes*** *represent the various types of actions that occur when a user interacts with components in a program. An event may occur when a user clicks a button, presses a key on the keyboard, selects an option from a menu, moves the mouse pointer, or performs any other interaction with a graphical component. Whenever such an action occurs, Java automatically creates an* ***event object*** *that describes the event. This object contains information about the event such as the source of the event, the time at which it occurred, and the type of interaction performed. The class used to define this object is called an* ***event class****.*
Event classes are primarily defined in the ***java.awt.event****. These classes form an important part of the Java event handling mechanism because they provide the necessary information required to process user actions.*

Concept of Event Classes
Conceptually, an event class can be understood as a ***representation of a user action within the program****. When a user performs an action, Java creates an instance of the corresponding event class and sends it to the event handling mechanism.*
This event object acts as a ***carrier of information*** *that helps the program understand what action occurred and where it occurred.*
The conceptual flow of an event can be represented as follows:
User Action

Event Occurs

Event Object Created

Event Listener Receives Event

Program Responds to Event
Thus, event classes act as ***information containers that describe user interactions with graphical components****.*

Common Event Classes
Java provides several event classes to represent different types of user interactions.
ActionEvent
An ActionEvent is generated when a user performs a general action on a component. This event commonly occurs when a button is clicked, a menu item is selected, or the Enter key is pressed in a text field.
MouseEvent

A MouseEvent is generated when the user interacts with the mouse. Examples include clicking the mouse button, pressing or releasing the mouse button, dragging the mouse, or moving the mouse pointer across the screen.

KeyEvent

A KeyEvent occurs when the user presses or releases a key on the keyboard. It is commonly used in applications that require keyboard input or keyboard shortcuts.

WindowEvent

A WindowEvent occurs when operations related to windows take place, such as opening, closing, minimizing, or activating a window.

ItemEvent

An ItemEvent occurs when the state of an item changes. For example, selecting or deselecting a checkbox or choosing an option in a combo box generates an item event.

Importance of Event Classes

Event classes are important because they provide detailed information about user interactions. Without event classes, a program would not be able to determine what type of action occurred or which component generated the event.

Each event object typically contains information such as:

- *The **source component** that generated the event*
- *The **type of event** that occurred*
- *The **time at which the event occurred***
- *Additional data related to the event*

This information allows the program to process events accurately and perform appropriate actions.

2.2 Source of Events

Introduction

*The **source of an event** refers to the object that generates the event when a user interacts with it. In graphical applications, the event source is usually a GUI component that the user can manipulate directly.*

*Whenever a user performs an action on a component, that component becomes the **event source**, and it generates an event object that describes the action.*

For example, when a user clicks a button, the button becomes the source of the event, and an ActionEvent is generated.

Concept of Event Source

*Conceptually, an event source can be understood as **the origin point from which an event is generated**. The source is responsible for creating the event and passing it to the event handling mechanism.*

The relationship between user actions and event sources can be represented as:

User Interaction

GUI Component (Event Source)

Event Generated

*Thus, the event source acts as the **starting point of the event handling process**.*

Common Event Sources

Several Swing components can act as event sources depending on the type of interaction performed by the user.

Some common event sources include:

- ***JButton** – generates ActionEvent when clicked*
- ***JTextField** – generates ActionEvent or KeyEvent when text is entered*
- ***JCheckBox** – generates ItemEvent when its state changes*
- ***JComboBox** – generates ActionEvent or ItemEvent when a selection changes*
- ***JList** – generates ListSelectionEvent when an item is selected*
- ***JMenuItem** – generates ActionEvent when selected*

Each of these components generates events according to the actions performed by the user.

2.3 Event Listener Interfaces

Introduction

*An **event listener interface** is used to detect and handle events generated by event sources. It contains methods that define the actions that should be performed when a specific event occurs.*

*Listener interfaces are also part of the **java.awt.event**. A class that wants to respond to events must implement the appropriate listener interface.*

When a listener is attached to an event source, it waits for events to occur. Once an event is generated, the listener receives the event object and executes the corresponding method.

Concept of Event Listener

*Conceptually, an event listener acts as **a receiver that listens for events and processes them**. It waits for a specific type of event and then executes the appropriate code when that event occurs.*

The relationship between the event source and event listener can be illustrated as:

Event Source

Event Generated

Event Listener Receives Event

Program Executes Action

This design allows Java programs to respond to user interactions in a structured and organized manner.

Common Listener Interfaces

Java provides many listener interfaces for handling different types of events.

ActionListener

Handles action events generated by components such as buttons or menu items.

MouseListener

Handles mouse events such as clicking, pressing, and releasing the mouse.

KeyListener

Handles keyboard events such as key presses and releases.

WindowListener

Handles events related to window operations such as opening or closing.

For example, the ActionListener interface contains the method:

actionPerformed(ActionEvent e)

This method is executed whenever an action event occurs.

2.4 Mouse Events

Introduction

Mouse events occur when a user interacts with a graphical component using a mouse device. These events allow the program to detect actions such as clicking, pressing, releasing, or moving the mouse pointer.

*Mouse events are represented by the **MouseEvent** class and are handled through the **MouseListener** interface.*

Types of Mouse Event Methods

The MouseListener interface provides several methods that respond to different mouse actions.

mouseClicked()

This method is executed when the user clicks a mouse button.

mousePressed()

This method is triggered when the mouse button is pressed down.

mouseReleased()

This method is triggered when the mouse button is released.

mouseEntered()

This method is called when the mouse pointer enters the area of a component.

mouseExited()

This method is called when the mouse pointer leaves the area of a component.

These methods enable programs to detect and respond to mouse-based interactions.

2.5 Keyboard Events

Introduction

Keyboard events occur when a user interacts with the keyboard by pressing, releasing, or typing keys. These events are essential in applications that require text input or keyboard-based control.
Keyboard events are represented by the ***KeyEvent*** *class and handled through the* ***KeyListener*** *interface.*

Types of Keyboard Event Methods
The KeyListener interface defines three important methods that respond to keyboard actions.
keyPressed()
This method is triggered when a key is pressed.
keyReleased()
This method is triggered when a key is released.
keyTyped()
This method is triggered when a key is typed, which usually occurs when a key is pressed and released.
These methods allow the program to capture and process keyboard input.
3. JavaFX
Introduction
JavaFX is a modern graphical user interface (GUI) framework used for developing rich and interactive desktop applications in Java. It provides a wide range of graphical components and tools that help programmers create visually attractive applications with advanced user interfaces.
JavaFX was introduced to overcome the limitations of earlier GUI technologies such as AWT and Swing. It supports modern interface design, animation effects, multimedia features, and flexible layouts. Because of these features, JavaFX is widely used for developing applications that require a sophisticated graphical environment.
One of the key concepts of JavaFX is the ***scene graph****, which represents the structure of graphical components in a hierarchical manner. In this structure, the main window contains a scene, and the scene contains various graphical elements such as buttons, text fields, and images.*
JavaFX also supports styling using CSS and interface design using FXML, which helps separate the user interface design from program logic.

3.1 Introduction to JavaFX
JavaFX is a software platform used to create graphical applications with rich user interfaces. It provides many built-in components such as buttons, labels, text fields, menus, tables, and charts that allow developers to design interactive applications easily.
The architecture of JavaFX is based on three main elements: ***Stage, Scene, and Nodes****.*
The ***Stage*** *represents the main window of the application. It is the top-level container that appears on the screen when the program runs.*
The ***Scene*** *acts as a container that holds all the graphical elements displayed in the window. It represents the content area of the application.*

*The **Nodes** are the individual graphical components placed inside the scene. Examples of nodes include buttons, labels, text fields, images, and shapes. Nodes can be organized in a hierarchical structure to form the graphical interface.*
This structure can be conceptually represented as follows:

```
Stage
 |
 └── Scene
      |
      ├── Button
      ├── Label
      ├── TextField
      └── Other UI Components
```

JavaFX also provides powerful features such as animation, multimedia playback, and graphical effects. These features make it possible to build modern and visually appealing applications.

3.2 JavaFX Setup

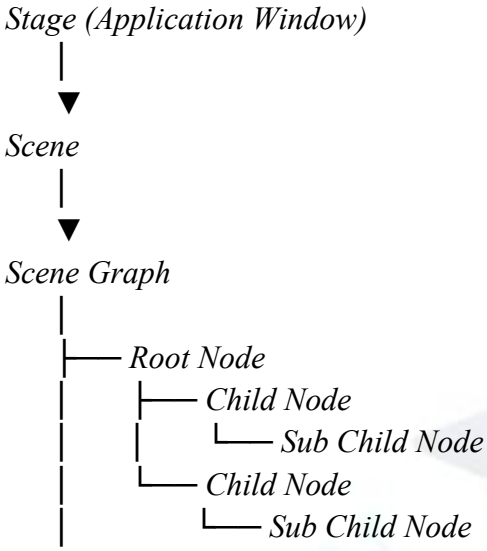
Before creating JavaFX applications, it is necessary to set up the required development environment. This setup ensures that the system has the necessary tools and libraries required to run JavaFX programs.
*The first step in setting up JavaFX is installing the **Java Development Kit (JDK)**, which provides the tools required to compile and run Java programs.*
Next, the JavaFX Software Development Kit (SDK) must be downloaded and installed. The JavaFX SDK contains the libraries required to create JavaFX applications.
After installing the JavaFX SDK, the development environment such as Eclipse, IntelliJ IDEA, or NetBeans must be configured to include the JavaFX libraries. This usually involves adding the JavaFX library files to the project settings so that the compiler can access them.
Once the setup process is complete, developers can create JavaFX programs and use various graphical components provided by the framework to build interactive user interfaces.
In summary, JavaFX setup involves installing the required software tools and configuring the development environment so that JavaFX libraries can be used in Java applications.

3.3 Scene Graph

Introduction

*In JavaFX, the **Scene Graph** is the fundamental structure used to represent and manage the graphical user interface of an application. It is a **hierarchical tree structure** that organizes all visual components that appear on the screen. Every graphical element such as buttons, labels, text fields, images, shapes, and layout containers is represented as a **node** in the scene graph.*
*The scene graph plays an essential role in JavaFX because it determines how components are **displayed, arranged, and interact with each other**. It allows developers to build complex user interfaces by organizing graphical elements in a structured manner.*

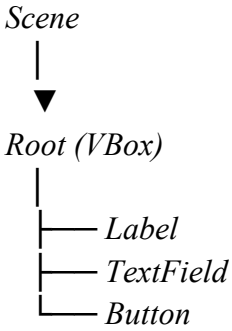
*In JavaFX applications, the **Stage** represents the main window, the **Scene** represents the content area within that window, and the **Scene Graph** represents the structure of components inside the scene.*
Conceptually, the relationship can be illustrated as follows:



Through this structure, JavaFX manages the layout, rendering, and behavior of graphical components.

3.3.1 Root

*The **Root node** is the topmost node in the scene graph and acts as the starting point of the entire graphical structure. Every JavaFX scene must have exactly **one root node**, which serves as the main container for all other nodes in the interface.*
*The root node usually represents a **layout container**, which is responsible for organizing and positioning child components. Common layout containers used as root nodes include **Pane, VBox, HBox, BorderPane, GridPane, and StackPane**.*
The root node controls how the child nodes are arranged within the scene. It defines the structure of the user interface and ensures that all components are placed correctly.
For example:



*In this structure, the **VBox acts as the root node**, and the label, text field, and button are placed inside it.*
*Because the root node is the highest level in the scene graph, it acts as the **foundation of the graphical interface**.*

3.3.2 Children

Children *are the nodes that exist directly under a parent node in the scene graph. These nodes represent the visible graphical components that appear within the user interface. Each parent node can contain multiple children nodes, and each child node may itself become a parent node if it contains additional components. This allows the interface to be organized in multiple levels.*

Examples of children nodes include:

- *Buttons*
- *Labels*
- *Text fields*
- *Images*
- *Shapes*
- *Layout containers*

Children nodes inherit certain properties from their parent nodes, such as visibility and transformations. They also follow the layout rules defined by the parent container.

Example:

Root (Pane)

```
|
├── Button
├── Label
└── TextField
```

In this case, the button, label, and text field are ***children of the root node****.*

Children nodes are responsible for displaying information and enabling user interaction with the application.

3.3.3 Hierarchy

The ***hierarchy*** *in a scene graph refers to the parent-child relationship between nodes that forms a tree-like structure. This hierarchical arrangement allows JavaFX to organize graphical elements efficiently and manage their relationships.*

In this hierarchy:

- *The* ***root node*** *is the top-level parent.*
- *Parent nodes can contain one or more* ***child nodes****.*
- *Child nodes can themselves become* ***parent nodes*** *if they contain other nodes.*

This layered structure makes it possible to create complex user interfaces by grouping related components together.

Example hierarchy:

Root (BorderPane)

```
|
├── Top (HBox)
|      ├── Button
|      └── Button
|
├── Center (VBox)
```

```
    ├── Label
    └── TextField
    |
    └── Bottom (HBox)
        └── Button
```

In this example:

- ***BorderPane** is the root node.*
- ***HBox and VBox** are intermediate parent nodes.*
- ***Buttons, labels, and text fields** are leaf nodes.*

This hierarchical structure provides several advantages:

- *It helps organize components logically.*
- *It controls the rendering order of graphical elements.*
- *It simplifies event handling and layout management.*

3.4 Layout Containers

Introduction

*In JavaFX, **layout containers** are special nodes that control the **arrangement, alignment, and positioning of graphical components within a scene**. When building graphical user interfaces, developers need to place components such as buttons, labels, text fields, images, and menus in an organized manner. Layout containers provide a structured way to manage these components automatically.*

*Instead of manually specifying the exact coordinates of each element on the screen, JavaFX uses layout containers that **automatically position components according to predefined layout rules**. This makes user interface design easier, more flexible, and responsive to changes in window size.*

*Layout containers are part of the **javafx.scene.layout** and are widely used in JavaFX applications. They act as **parent nodes in the scene graph**, while the graphical components inside them are considered **child nodes**.*

By using layout containers, developers can create well-structured interfaces that adapt automatically when the window is resized or when new components are added.

*Common layout containers used in JavaFX include **VBox, HBox, GridPane, and StackPane**.*

3.4.1 VBox

Introduction

***VBox (Vertical Box)** is a layout container that arranges its child nodes **vertically in a single column**. All components placed inside a VBox are arranged from top to bottom in the order in which they are added.*

VBox is commonly used when the interface requires components to appear in a vertical sequence, such as forms, menus, or lists of options.

Concept of VBox

Conceptually, VBox organizes elements in a vertical stack.

VBox

```
|
├── Label
```

├── TextField
├── Button
└── Button

*Each component is placed **below the previous component**, forming a vertical arrangement.*

Characteristics of VBox

- *Arranges components **vertically from top to bottom**.*
- *Maintains the **order in which components are added**.*
- *Allows control over **spacing between components**.*
- *Supports alignment options such as **center, left, or right alignment**.*
- *Automatically adjusts component placement when the window size changes.*

*VBox is widely used in **login forms, settings panels, and vertical navigation menus**.*

3.4.2 HBox

Introduction

***HBox (Horizontal Box)** is a layout container that arranges its child nodes **horizontally in a single row**. Components placed inside an HBox appear side by side from left to right.*

HBox is commonly used when multiple components must be displayed in a horizontal arrangement.

Concept of HBox

Conceptually, HBox arranges components in a horizontal sequence.

HBox

│
├── Button
├── Button
├── Button
└── Button

*Each component is placed **next to the previous component**.*

Characteristics of HBox

- *Arranges components **horizontally from left to right**.*
- *Maintains the order of components.*
- *Supports spacing between components.*
- *Allows alignment options such as center or right alignment.*
- *Automatically adjusts layout when the window size changes.*

*HBox is commonly used for **toolbars, button groups, and navigation bars**.*

3.4.3 GridPane

Introduction

***GridPane** is a layout container that organizes components in a **grid structure consisting of rows and columns**. Each component is placed in a specific cell defined by row and column indices.*

*GridPane is particularly useful when designing structured interfaces such as **forms, tables, and data entry layouts**.*

Concept of GridPane

Conceptually, GridPane divides the interface into a grid.
GridPane

```
 --------------------------------
| Label | TextField            |
| Label | TextField            |
|       | Button               |
 --------------------------------
```

*Each component is positioned using **row and column coordinates**.*

Characteristics of GridPane

- *Organizes components into **rows and columns**.*
- *Allows precise placement of components in grid cells.*
- *Supports **row and column spanning**, allowing components to occupy multiple cells.*
- *Suitable for complex user interfaces requiring structured alignment.*

*GridPane is widely used in **registration forms, login screens, and data entry systems**.*

3.4.4 StackPane

Introduction

***StackPane** is a layout container that places all child nodes **on top of each other in a stack-like arrangement**. Components share the same space, and the most recently added component appears above the others.*

StackPane is useful when components need to overlap or when one component acts as a background for others.

Concept of StackPane

Conceptually, StackPane layers components.
StackPane

```
   |
   ├── Background Image
   ├── Label
   └── Button
```

All components occupy the same area but appear in different layers.

Characteristics of StackPane

- *Places components **on top of each other**.*
- *Supports alignment within the stack (center, top, bottom).*
- *Useful for **overlay layouts and background elements**.*
- *Simplifies designs where components must be centered.*

*StackPane is commonly used for **background images, centered messages, and overlay interfaces**.*

3.5 Buttons and Controls

Introduction

*In JavaFX, **buttons and controls** are graphical components that allow users to interact with an application. These components form an important part of the user interface because they enable users to perform actions, enter information, and select options within a program.*

Controls are interactive elements that respond to user actions such as clicking a button, typing text, selecting items from a list, or choosing options from menus. JavaFX provides a wide range of built-in controls that help developers design rich and interactive graphical user interfaces.
These controls are mainly available in the ***javafx.scene.control****. By using these controls, developers can create applications that allow users to communicate with the program in a simple and intuitive manner.*

Concept of Buttons and Controls
Conceptually, ***buttons and controls represent interactive nodes in the JavaFX scene graph that allow communication between the user and the application****. When a user interacts with these controls, events are generated and the application responds accordingly.*
For example:
User Interaction
↓
JavaFX Control
↓
Event Generated
↓
Program Executes Action
Thus, buttons and controls act as the ***interface through which users send commands to the program****.*

Common JavaFX Controls
JavaFX provides several types of controls that are commonly used in graphical applications.
Button
A button is a control that performs a specific action when clicked by the user. It is commonly used to submit data, start operations, or trigger commands in an application.
Example concept:
[Submit] [Cancel] [Login]

Label
A label is used to display ***text or images*** *in a graphical interface. Labels are mainly used to provide information or describe other controls such as text fields.*
Example:
Username:
Password:

TextField

*A TextField is used to accept **single-line text input** from the user. It allows users to enter information such as names, numbers, or short messages.*
Example:
Name: [_____]

PasswordField
*A PasswordField is used to enter **passwords securely**. The characters typed by the user are hidden and replaced with symbols such as asterisks (*) to maintain privacy.*
Example:
*Password: ********

CheckBox
A CheckBox allows users to select or deselect an option. It is often used when multiple selections are allowed.
Example:
☐ Accept Terms and Conditions

RadioButton
*Radio buttons allow the user to select **only one option from a group of choices**. When one radio button is selected, the others in the group are automatically deselected.*
Example:
( ) Male
( ) Female

ComboBox
*A ComboBox provides a **drop-down list of options** from which the user can select one item.*
Example:
Select Country
[India ▼]

Importance of Buttons and Controls
Buttons and controls are essential for creating interactive graphical interfaces. They allow users to perform operations, enter data, and navigate through the application.
Without these components, applications would not be able to respond to user input effectively.
*Buttons and controls therefore serve as the **primary communication mechanism between the user and the software application**.*

3.6 Styling in JavaFX

Introduction
*In JavaFX applications, **styling** refers to the process of defining and controlling the **visual appearance of graphical components** such as buttons, labels, text fields, and layout containers. Styling determines how these components look on the screen, including their colors, fonts, borders, backgrounds, spacing, and alignment.*

A well-designed user interface is important for improving the usability and attractiveness of an application. JavaFX provides powerful styling capabilities that allow developers to customize the appearance of components and create visually appealing interfaces.
*One of the major advantages of JavaFX is that it supports **CSS (Cascading Style Sheets)** for styling. This allows developers to separate the **presentation layer** (design and appearance) from the **application logic**. As a result, the interface can be modified without changing the program code.*
*Styling in JavaFX works by applying style properties to nodes in the **scene graph**, which control how those nodes are rendered on the screen.*

Concept of Styling in JavaFX

*Conceptually, styling in JavaFX involves applying **visual design rules** to graphical components in order to enhance their appearance and usability.*
*In a JavaFX application, every graphical component is represented as a **node** within the scene graph. Each node has properties that control its appearance. Styling modifies these properties to create different visual effects.*
The conceptual flow of styling can be represented as follows:
JavaFX Node (Button, Label, TextField)

Style Properties Applied

Visual Appearance Modified

User Interface Displayed
*Through styling, developers can create user interfaces that are **modern, consistent, and visually attractive**.*

JavaFX CSS Styling

*JavaFX uses a styling system similar to **web-based CSS**, which allows developers to define style rules for graphical components. These rules specify how components should appear and behave visually.*
*In JavaFX CSS, each component can be styled using **selectors** and **properties**.*
For example, a button can be styled using properties that control its background color, font size, text color, and border.
Example CSS rule:

```
.button {
    -fx-background-color: blue;
    -fx-text-fill: white;
    -fx-font-size: 14px;
}
```

This rule changes the appearance of all buttons by setting the background color to blue, the text color to white, and the font size to 14 pixels.

Methods of Applying Styles in JavaFX

JavaFX allows styles to be applied in several ways.

Inline Styling

Inline styling is applied directly within the Java code using the setStyle() method. In this approach, style properties are written as a string inside the program.

Example concept:

Button

└── setStyle("-fx-background-color: green; -fx-text-fill: white;");

Inline styling is useful for quick styling but is not recommended for large applications because it mixes design with program logic.

External CSS Files

External CSS files provide a better approach for styling JavaFX applications. In this method, all style rules are written in a separate CSS file and then applied to the application.

This allows developers to maintain the design separately from the program code.

Conceptual structure:

JavaFX Application

├── Java Code

└── CSS File

└── Style Rules

*External CSS styling improves **code organization, maintainability, and flexibility**.*

Styling Using Style Classes and IDs

*JavaFX components support **style classes** and **ID selectors** for applying styles to specific components.*

Style Class

A style class allows multiple components to share the same styling rule.

Example:

.button

This selector applies styles to all button components.

ID Selector

An ID selector applies styles to a specific component.

Example:

#loginButton

This rule styles only the component whose ID is loginButton.

*Using style classes and IDs helps developers create **precise and reusable styling rules**.*

Common JavaFX Style Properties

JavaFX CSS provides many style properties for customizing components.
Some commonly used properties include:

- ***-fx-background-color*** *– sets the background color of a component*
- ***-fx-text-fill*** *– controls the color of text*
- ***-fx-font-size*** *– sets the size of the font*
- ***-fx-font-family*** *– defines the font type*
- ***-fx-border-color*** *– sets the border color*
- ***-fx-padding*** *– controls the space around components*
- ***-fx-alignment*** *– determines the alignment of content*

These properties help developers design visually consistent user interfaces.

Importance of Styling in JavaFX

Styling plays an important role in JavaFX application development because it enhances both ***usability and visual appeal****.*
Through styling, developers can:

- *Improve the visual appearance of user interfaces*
- *Maintain consistent design across the entire application*
- *Separate design from program logic*
- *Easily update the interface without modifying program code*

Well-designed styling makes applications easier to use and more attractive for users.

3.7 Animation and Transitions in JavaFX

Introduction

In graphical user interface applications, visual movement and dynamic effects improve the overall user experience. ***Animation in JavaFX*** *refers to the process of creating movement or visual change in graphical components over a period of time. By using animation, developers can make components such as buttons, images, or shapes move, rotate, fade, or change size gradually.*
JavaFX provides powerful tools for creating animations and transitions in applications. These tools allow developers to build interfaces that are more interactive, attractive, and user-friendly. Animation is widely used in modern software to provide smooth visual feedback when users perform actions.
JavaFX supports animation through classes available in the ***javafx.animation****. These classes allow developers to define how the properties of graphical components change with time.*

Concept of Animation in JavaFX

Conceptually, animation is created by ***modifying the properties of a graphical object continuously over time****. In JavaFX, every graphical element is represented as a* ***node*** *in the scene graph. Animation works by changing properties of these nodes, such as position, rotation, size, or opacity.*

For example, a graphical object may move from one position to another, gradually change its color, or rotate continuously. These changes occur in small steps over time, which creates the visual illusion of movement.

The general process of animation can be represented as:

Node (Button, Image, Shape)

Property Change Over Time

Animation Effect

Through this mechanism, JavaFX enables developers to create smooth transitions and dynamic user interfaces.

Animation Mechanism in JavaFX

*JavaFX animations are usually based on **time-based changes** in the properties of nodes. The framework automatically updates the graphical components at specific intervals so that the movement appears smooth.*

Animations in JavaFX are generally controlled using two main concepts:

- ***Timeline animations***
- ***Transition animations***

Both approaches allow developers to define how and when graphical properties change.

Timeline Animation

*The **Timeline** class is used to create animations by specifying a sequence of **key frames**. A key frame represents a point in time at which a particular property value is defined.*

In timeline animation, developers specify:

- *The property that should change*
- *The starting value of the property*
- *The final value of the property*
- *The duration of the animation*

During the animation process, JavaFX gradually changes the property from the starting value to the final value.

Conceptually:

Time 0s → Time 2s → Time 4s

Value A Value B Value C

The Timeline class is useful when complex animations require precise control over multiple properties and time intervals.

Transition Animations

*JavaFX also provides **transition classes**, which simplify the creation of animations. Instead of defining detailed property changes manually, developers can use predefined animation classes.*

These classes automatically handle the animation process and allow developers to specify only the important parameters such as duration and target node.
Several transition classes are available in JavaFX.

FadeTransition

*The **FadeTransition** class is used to create a fading effect by changing the **opacity** of a node.*
Opacity determines how visible or transparent an object is.
In a fade animation, the object may gradually become invisible or gradually appear on the screen.
Example concept:
Opacity 1.0 → Opacity 0.5 → Opacity 0.0
Fully Visible Partially Visible Invisible
*Fade transitions are often used for **smooth appearance or disappearance of interface elements**.*

TranslateTransition

*The **TranslateTransition** class is used to move a node from one position to another along the horizontal or vertical direction.*
This animation modifies the position of a node by changing its translation properties.
Example concept:
Start Position → Intermediate Position → Final Position
*Translate transitions are useful for **sliding menus, moving images, or shifting interface components**.*

RotateTransition

*The **RotateTransition** class rotates a node around its center or a specified axis.*
The rotation occurs gradually during the animation period.
Example concept:
0° → 90° → 180° → 360°
*Rotate transitions are commonly used for **spinning icons, loading indicators, or rotating objects**.*

ScaleTransition

*The **ScaleTransition** class changes the **size of a node** by scaling it along the horizontal and vertical directions.*
This animation can make objects appear larger or smaller.
Example concept:
Small Object → Medium Size → Large Object
*Scale transitions are often used for **zoom effects and emphasis on important components**.*

PathTransition

*The **PathTransition** class allows a node to move along a **specified path**. The path can be a line, curve, circle, or any geometric shape.*

Instead of moving in a straight line, the node follows the shape defined by the path.
Example concept:
Node Movement Along Path
Start → Curve → End
This animation is useful for creating ***complex motion patterns****.*

Importance of Animation in User Interfaces
Animation is an important feature in modern graphical applications because it improves interaction between the user and the application.
Animations provide visual feedback when users perform actions. For example, a button may change appearance when clicked, or a menu may slide smoothly into view. These effects make the interface easier to understand and more enjoyable to use.
Animation also helps guide the user's attention toward important elements of the interface.

Advantages of Animation in JavaFX
Animation in JavaFX offers several advantages.

- *It enhances the* ***visual appearance of applications****.*
- *It improves* ***user interaction and feedback****.*
- *It allows smooth transitions between interface elements.*
- *It helps create* ***modern and professional user interfaces****.*

Smart Study PLUS