

JAVA

Programming

Think in Java

<by Sumit/>

Mr. Sumit

Associate Professor, Brahmanand Institute of Teachers Education

Chaparda Junagadh

Gujarat

Email ID: sumit.bknmu@gmail.com

YT: @bcaschool

Insta: @sumitwanza

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from the Brahmanand Institute of Teachers Education.

Java is one of the most widely used programming languages in the world, known for its versatility, efficiency, and platform independence. This book is crafted to introduce Java programming in a clear and structured manner, guiding you from the basics to more advanced topics with hands-on examples.

The book is organized into five units, each building upon the concepts introduced in the previous one. The first three units cover essential topics for all Java developers, starting with basic syntax, data types, and control structures, followed by object-oriented programming concepts such as classes, inheritance, and polymorphism. These foundational concepts will equip you with the skills needed to write functional Java programs.

The fourth unit takes a deeper dive into more advanced features of Java, such as collections, exception handling, and file I/O. These topics form the backbone of robust Java applications and will require practice and experimentation. I encourage you to attempt the exercises before consulting the solutions to deepen your understanding.

The fifth and final unit introduces Java's capabilities for building dynamic web applications using frameworks like Spring and JavaFX. While the configuration and setup may seem challenging at first, the satisfaction of creating your first web application will make the effort worthwhile.

All programs discussed in this book are available on the dedicated book webpage. I recommend following a three-step approach:

Write the code yourself,

Review the provided solution, and

Modify and extend the code to explore new possibilities.

Java's extensive libraries and frameworks continuously evolve, providing new ways to solve complex problems. The goal of this book is not only to teach you the fundamentals but also to ignite your passion to continue learning and growing as a developer. I hope you find the examples and explanations in this book both practical and inspiring. Should you have any suggestions or feedback to improve the content, feel free to reach out directly.

Sumit Wanza

Guidelines for Students

Start with the Basics

Begin with a solid understanding of the basic syntax and constructs of Java. Don't rush through the foundational concepts, as they are crucial for your success with more complex topics later.

Understand the Concepts

Focus on understanding the "why" behind the code, not just the "how." Grasping the logic and principles behind Java's constructs will make it easier to apply them in different situations.

Practice Regularly

Programming is a skill that improves with practice. After reading each chapter, work on the examples and exercises provided. Try to write the code on your own before referring to the solutions. The more you practice, the better you'll understand the concepts.

Experiment and Extend

The examples provided in this book are just the starting point. After studying each example, try modifying it, adding new features, or creating your own variations. Experimenting with code is one of the best ways to learn.

Focus on Problem-Solving Skills

While learning Java syntax is important, developing strong problem-solving skills is essential. Learn how to approach problems logically, break them down into manageable parts, and then write code to solve them.

CONTENTS

UNIT 1: JAVA OVERVIEW	
1.1 History and Features of Java	3
1.2 JDK, JVM and JRE	3
1.3 JDK Tools	4
1.4 JIT compiler	5
1.5 Java IDE (NetBeans and Eclipse)	6
1.6 Byte code and machine code	7
Basic building blocks:	
1.7 Data Type (Integer, Float, Character, Boolean)	8
1.8 Java Tokens	10
1.9 Operators	17
Basics of Java programming	
2.0 Writing, Compiling and Executing basic Java Program	26
2.1 Java variable and scope of variables	18
2.2 Decision Statements (if, switch)	27
2.3 Looping Statements (for, while, Do-While)	30
2.4 Jumping Statements (break, continue, return)	30
2.5 Array (One Dim., Rectangular, Jagged)	23
2.6 Command Line Argument	25
OOP Concepts	
2.7 Definition and concepts of Class, Object, Encapsulation, Inheritance, Polymorphism	
2.8 Creating and using Class with members	
2.9 Static and Non-Static Members	
3.0 Constructor	35
3.1 finalize () method	37
3.2 Constructor overloading and Method overloading	41
UNIT 2: Inheritance	43
3.3 Definition, concept and types	
3.4 Universal Class (Object Class)	
3.5 Access Specifiers (public, private, protected, default, private protected)	49
3.6 Constructors in inheritance	
3.7 Method Overriding	
Interface	
3.8 Definition and concept	
3.9 Defining Multiple Interface in a single program	

4.0 Execution of interface in simple application	
4.1 Implementing Hybrid Inheritance program using Interface	
4.2 Nested and Inner Class	
4.3 Abstract and Final Class	
4.4 Normal import and Static Import	
Java API packages and important classes:	
4.5 Introduction to API packages	
4.6 Various packages (java.lang, java.util, java.io, java.net, java.awt, java.awt.event, java.applet, java.swing)	
4.7 java.lang Package Classes (Math, Wrapper Classes, String, String Buffer, Number, Package)	
4.8 java.util Package Classes (Random, Date, GregorianCalendar, Vector, HashTable, StringTokenizer, SimpleTimeZone, Stack, Scanner)	
4.9 Creating and Using User Defined package and sub-package	
Exception Handling	
5.0 Definition and concept	
5.1 Try, catch, finally, throw, throws	
5.2 Creating user defined Exception class	
UNIT 3: Threads:	
5.3 Definition, concept and life cycle (Various states)	
5.4 Thread Class and its methods	
5.5 Multithreading implementation and Synchronization	
5.6 Deamon Thread, Non-Deamon Thread	
Stream	
5.7 Introduction	
5.8 Types (Input, Output, Character, Byte)	
Applets	
5.9 Introduction to Applet	
6.0 Applet Life Cycle	
6.1 Implement & Executing Applet with Parameters Playing Audio Example	
6.2 Graphics class	
Layout Managers	
6.3 Flow-Layout	
6.4 Border-Layout	
6.5 Card-Layout	
6.6 Grid-Layout	
6.7 Grid-Bag Layout	
6.8 Introduction to other layouts(Box-Layout, Spring-Layout, Group-Layout)	
6.9 Using NO LAYOUT Manager	
UNIT 4: Creating GUI using swing	
7.0 Introduction to AWT and Swing	

7.1 Difference Between AWT and Swing	
Swing Components	
7.2 JFrame, JPanel, JLabel, JButton, JRadioButton, JCheckBox, JPasswordField, JTextArea, JScrollBar, JComboBox, JList, JToggleButton, JTabbedPane, JSlider, JProgressBar, JTextPane	
7.3 Menus (JMenuBar, JMenu, JMenuItem)	
Event handling	
7.4 Introduction and concept	
7.5 Event Delegation Model	
7.6 Event Packages (AWT Event Package, Swing Event Package)	
7.7 Event Classes (ActionEvent, ItemEvent, FocusEvent, MouseEvent, MouseWheelEvent, TextEvent)	
7.8 Listener Interfaces (ActionListener, ItemListener, FocusListener, KeyListener, MouseListener, MouseMotionListener, TextListener, WindowListener)	
7.9 Adaptor Classes (FocusAdaptor, KeyAdaptor, MouseAdaptor, MouseMotionAdaptor)	

HOW TO INSTALL JDK FOR JAVA DEVELOPMENT:

Java Development Kit (JDK) is essential for writing, compiling, and running Java programs. Follow these steps to install JDK on your system:

Step 1: Download the JDK

1. Visit the official Oracle website:
<https://www.oracle.com/java/technologies/javase-downloads.html>.
2. Select the version of JDK suitable for your operating system (Windows, macOS, or Linux).
3. Click the download link and accept the license agreement.

Step 2: Install the JDK

1. Locate the downloaded installer file in your system's downloads folder.
2. Double-click the installer to begin the installation.
3. Follow the on-screen instructions:
 - Select the installation path (default path is usually fine).
 - Complete the setup by clicking "Next" or "Install".

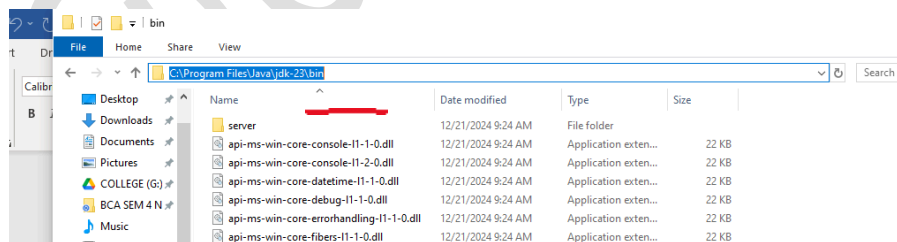
Step 3: Set Up Environment Variables (Windows Only)

1. Open **System Properties** by searching for "Environment Variables" in the Start Menu.
2. In the **System Properties** window, click on the **Environment Variables** button.
3. Under **System Variables**, find and select the Path variable, then click **Edit**.
4. Add the path to the JDK's bin directory (e.g., C:\Program Files\Java\jdk-XX.X.X\bin) and click **OK**.
5. To verify the setup, open a command prompt and type: `java -version` You should see the installed JDK version.

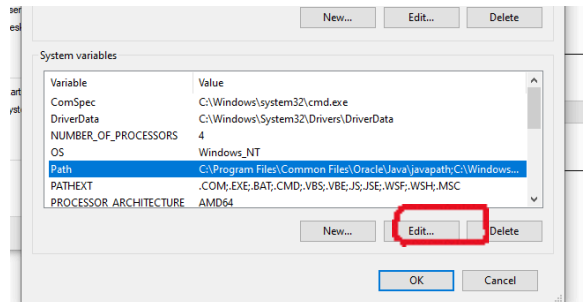
Step 4: Verify Installation

1. Open a terminal or command prompt.
2. Type the following command to check the Java Compiler: `javac -version` This confirms that the JDK is successfully installed.

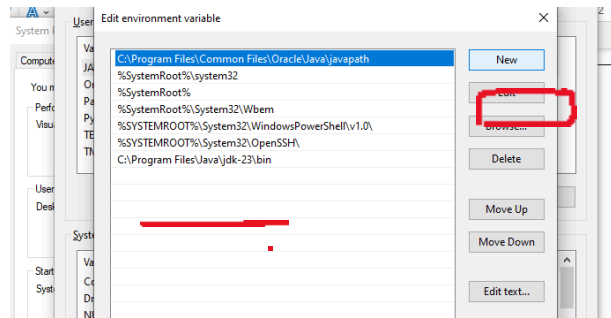
- Open Java folder by default is: C:\Program Files\Java\jdk-23
- Copy the bin folder path



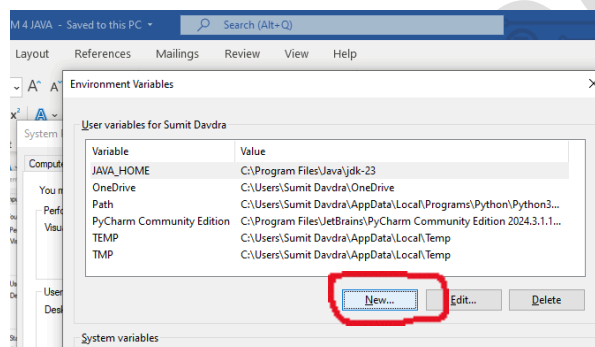
- Open Edit the system Environment variable.
- Then click Environment variables button.



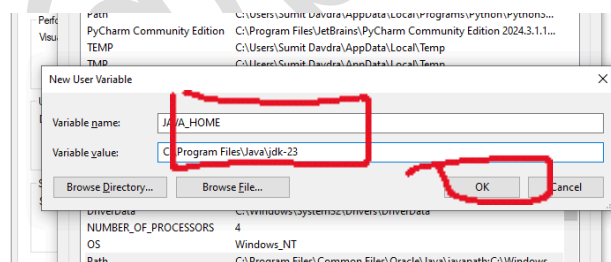
- Select Path and hit edit button



- Click New Button and paste the copied path. (C:\Program Files\Java\jdk-23\bin) and press OK button.
- Come previously tab and click New Button.



- Now set this name and path as per image.



- Now press all ok & apply button and open CMD to check Java properly installed and path set.
- Open CMD and type java -version and hit enter, enjoy java.

UNIT 1: OVERVIEW

HISTORY AND FEATURES OF JAVA

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]). History of even naming of the Java is very interesting. It went under many names.

GreenTalk

James Gosling was leading a team named as 'Green' team. Target of this team was to create a new project which can. Initially C++ was the original choice to develop the project. James Gosling wanted to enhance C++ to achieve the target but due to high memory usage, that idea was rejected and team started with a new language initially named as GreenTalk. The file extension used as .gt. Later this language was termed as Oak and finally to Java.

Oak

James Gosling renamed language to Oak. There was an Oak tree in front of his office. James Gosling used this name as Oak represents solidarity and Oak tree is the national tree of multiple countries like USA, France, Romania etc. But Oak technologies already had Oak as a trademark and James team had to brainstorm another title for the language.

Java

Team put multiple names like DNA, Silk, Ruby and Java. Java was finalized by the team. James Gosling tabled Java title based on type of espresso coffee bean. Java is an island in Indonesia where new coffee was discovered termed as Java coffee. As per James Gosling, Java was among the top choice along with Silk. Finally, Java was selected as it was quite unique and represented the essence of being dynamic, revolutionary and fun to say.

WHAT IS JDK?

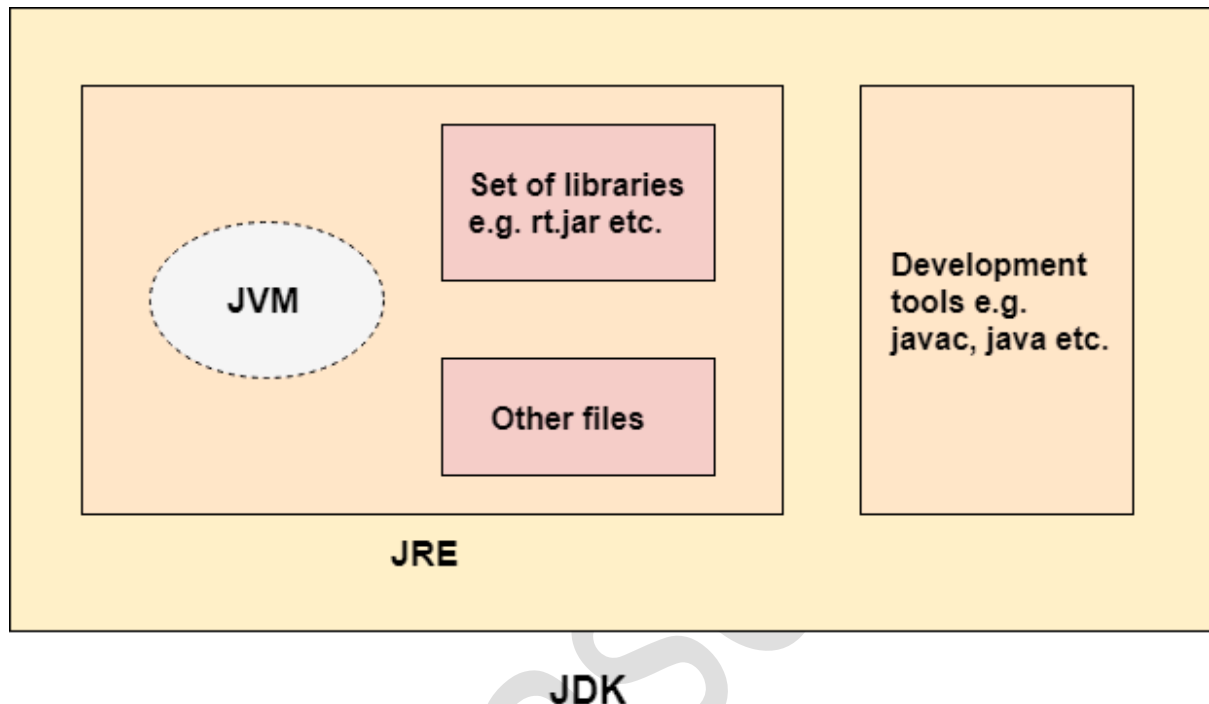
JDK is an abbreviation for Java Development Kit which includes all the tools, executables, and binaries required to compile, debug, and execute a Java Program. JDK is platform dependent i.e. there are separate installers for Windows, Mac, and Unix systems. JDK includes both JVM and JRE and is entirely responsible for code execution. It is the version of JDK that represents a version of Java.

WHAT IS JRE?

JRE is a Java Runtime Environment which is the implementation of JVM i.e. the specifications that are defined in JVM are implemented and create a corresponding environment for the execution of code. JRE comprises mainly Java binaries and other classes to execute the program like JVM which physically exists. Along with Java binaries JRE also consists of various technologies of deployment, user interfaces to interact with code executed, some base libraries for different functionalities, and language and util-based libraries.

WHAT IS JVM?

JVM is the abbreviation for Java Virtual Machine which is a specification that provides a runtime environment in which Java byte code can be executed i.e. it is something that is abstract and its implementation is independent of choosing the algorithm and has been provided by Sun and other companies. It is JVM which is responsible for converting Byte code to machine-specific code. It can also run those programs which are written in other languages and compiled to Java bytecode. The JVM performs the mentioned tasks: Loads code, Verifies code, Executes code, and Provides runtime environment.



JDK TOOLS

JDK (Java Development Kit) provides a collection of tools and utilities that assist developers in building, testing, and running Java programs. These tools are typically grouped into development, diagnostic, monitoring, and deployment tools. Below is a brief overview of some key JDK tools:

Development Tools

- **javac:** The Java Compiler - Converts Java source code (.java) into bytecode (.class).
- **java:** The Java Launcher - Executes Java applications by launching the Java Virtual Machine (JVM).
- **javadoc:** Generates HTML documentation from Java source code comments.
- **jar:** Java Archive Tool - Packages multiple files into a .jar archive.
- **jdb:** Java Debugger - A command-line tool for debugging Java applications.
- **javap:** The Java Class File Disassembler - Provides details about compiled class files.

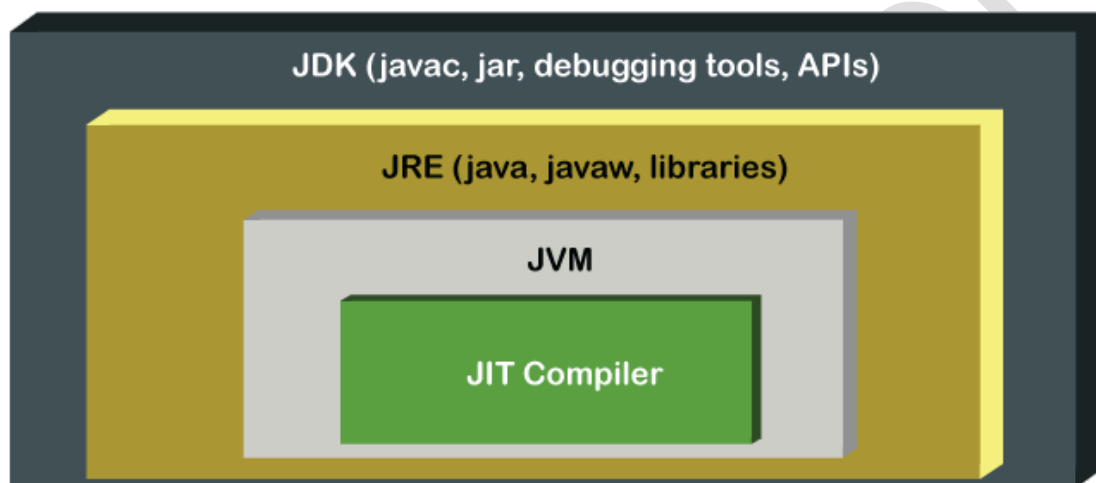
JIT IN JAVA

When we write a program in any programming language it requires converting that code in the machine-understandable form because the machine only understands the binary language. According to the programming languages, compiler differs. The compiler is a program that converts the high-level language to machine level code. The Java programming language uses the compiler

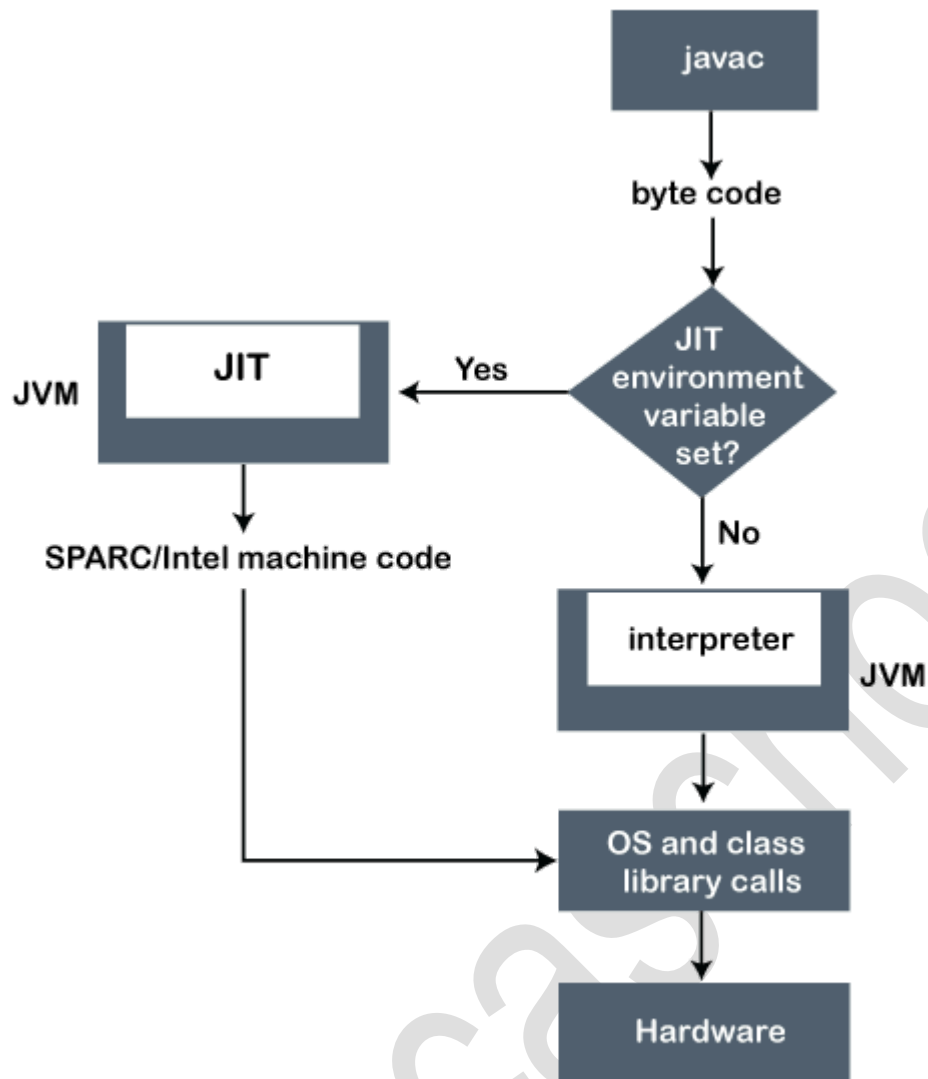
named javac. It converts the high-level language code into machine code (bytecode). JIT is a part of the JVM that optimizes the performance of the application. JIT stands for Java-In-Time Compiler. The JIT compilation is also known as dynamic compilation. In this section, we will learn what is JIT in Java, its working, and the phases of the JIT compiler.

WHAT IS JIT IN JAVA?

JIT in Java is an integral part of the JVM. It accelerates execution performance many times over the previous level. In other words, it is a long-running, computer-intensive program that provides the best performance environment. It optimizes the performance of the Java application at compile or run time.



JIT compiler boosts the performance of the native application. We can understand the working of the JIT compiler with the help of the following flow chart.



JIT Compilation Process

JAVA IDE

A Java Integrated Development Environment (IDE) is a software application that provides developers with a comprehensive environment for Java development. It simplifies the coding process with features like code completion, debugging tools, and integrated build systems. Here are some popular Java IDEs:

ECLIPSE

Overview: A widely used, open-source IDE with extensibility via plugins.

Features:

- Supports Java EE development with additional plugins.
- Built-in debugger and profiler.
- Integration with version control systems like Git.
- Marketplace for installing add-ons.
- Best For: Developers looking for a free, customizable IDE with broad community support.

NETBEANS (APACHE)

Overview: An open-source IDE officially supported by Oracle for Java.

Features:

- Simplified project management with built-in support for Maven and Gradle.
- Tools for debugging and profiling.
- Support for web and enterprise development (Java EE).
- Best For: Beginners and those who prefer a straightforward setup.

BYTECODE

Bytecode is an intermediate representation of a program that lies between the source code and machine code. It is generated by compilers for certain programming languages, such as Java or Python, and is designed to be executed by a virtual machine (e.g., the Java Virtual Machine or Python interpreter).

Key Features of Bytecode:

- Platform Independence:
- Efficiency
- Security
- Portability
- When a Java program is compiled, the .java file is converted into a .class file containing bytecode. The Java Virtual Machine (JVM) then executes this bytecode on the target machine.
 - It technically a type of machine code. but it is not binary
 - is machine code built for the JAVA VIRTUAL MACHINE (JVM)
 - is the .class file that was compiled from java code using the command
 - javac program.java
 - can be run on any machine that has the JVM installed using the command
 - java program

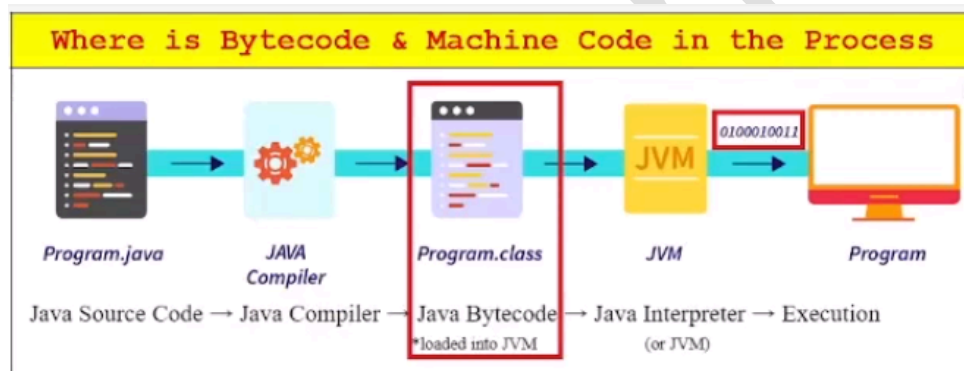
MACHINE CODE

Machine code is the lowest-level programming language, consisting of binary instructions (0s and 1s) that a computer's central processing unit (CPU) can directly execute. It represents the fundamental operations that the hardware can perform, such as arithmetic, data movement, and logic operations.

- the lowest level of code. Also Known As binary code
- syntax of the machine code varies depending on CPU and computer's Operating system

BINARY CODE

- is the ones and zeros that can be read by the central processing unit (CPU) of the machine, hence the name machine language.



BASIC BUILDING BLOCKS:

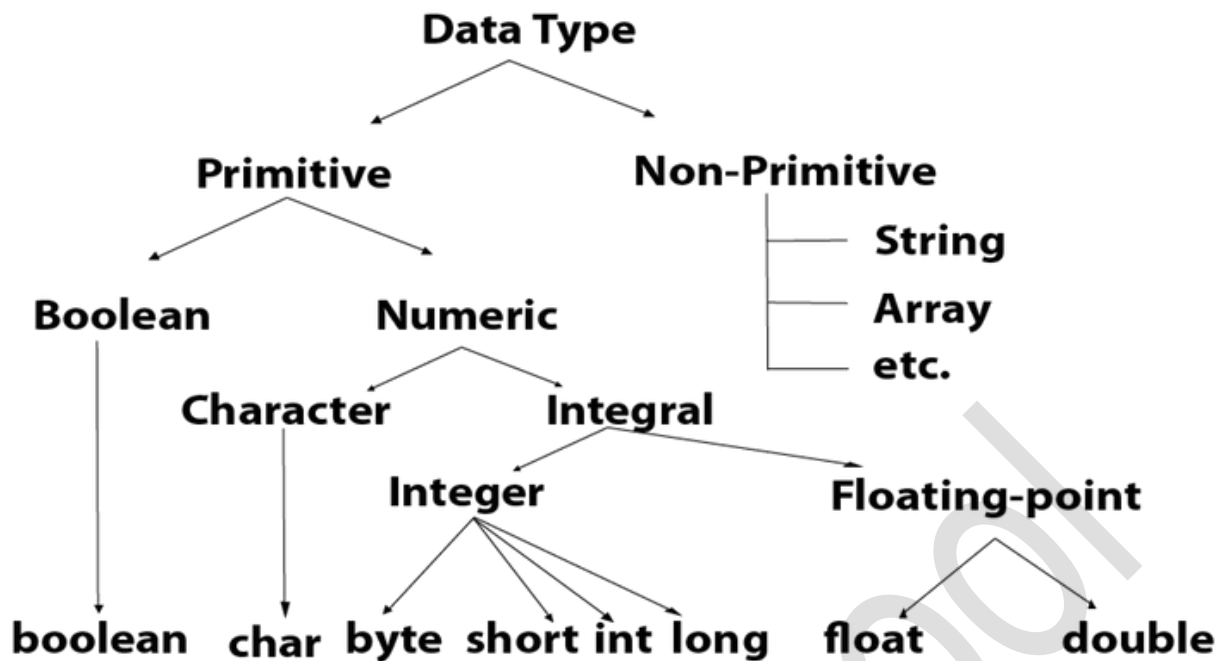
Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

- Primitive data types: The primitive data types include Boolean, char, byte, short, int, long, float and double.
- Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.



BOOLEAN DATA TYPE

In Java, the boolean data type represents a single bit of information with two possible states: true or false. It is used to store the result of logical expressions or conditions. Unlike other primitive data types like int or double, boolean does not have a specific size or range. It is typically implemented as a single bit, although the exact implementation may vary across platforms.

Example:

```
Boolean a = false;
```

```
Boolean b = true;
```

INT DATA TYPE

The int data type in Java is a primitive data type that represents a 32-bit signed two's complement integer. It has a range of values from -2,147,483,648 to 2,147,483,647. The int data type is one of the most commonly used data types in Java and is typically used to store whole numbers without decimal points. Its default value is 0.

```
int a = 100000, int b = -200000
```

FLOAT DATA TYPE

The float data type in Java is a primitive data type that represents single-precision 32-bit IEEE 754 floating-point numbers. It can represent a wide range of decimal values, but it is not suitable for precise values such as currency. The float data type is useful for applications where a higher range of values is needed, and precision is not critical.

```
float f1 = 234.5
```

CHAR DATA TYPE

The char data type in Java is a primitive data type that represents a single 16-bit Unicode character. It can store any character from the Unicode character set, that allows Java to support internationalization and representation of characters from various languages and writing systems.

```
char letterA = 'A'
```

The char data type is commonly used to represent characters, such as letters, digits, and symbols, in Java programs. It can also be used to perform arithmetic operations, as the Unicode values of characters can be treated as integers. For example, you can perform addition or subtraction operations on char variables to manipulate their Unicode values.

JAVA TOKENS

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

List of Java Keywords:

1. **abstract:** Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.
2. **boolean:** Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. **break:** Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. **byte:** Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. **case:** Java case keyword is used with the switch statements to mark blocks of text.
6. **catch:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. **char:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. **class:** Java class keyword is used to declare a class.
9. **continue:** Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. **default:** Java default keyword is used to specify the default block of code in a switch statement.
11. **do:** Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.
12. **double:** Java double keyword is used to declare a variable that can hold 64-bit floating-point number.
13. **else:** Java else keyword is used to indicate the alternative branches in an if statement.
14. **enum:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. **extends:** Java extends keyword is used to indicate that a class is derived from another class or interface.
16. **final:** Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.
17. **finally:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.
18. **float:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.

19. for: Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.
20. if: Java if keyword tests the condition. It executes the if block if the condition is true.
21. implements: Java implements keyword is used to implement an interface.
22. import: Java import keyword makes classes and interfaces available and accessible to the current source code.
23. instanceof: Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. int: Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. interface: Java interface keyword is used to declare an interface. It can have only abstract methods.
26. long: Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. native: Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. new: Java new keyword is used to create new objects.
29. null: Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. package: Java package keyword is used to declare a Java package that includes the classes.
31. private: Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. protected: Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.
33. public: Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. return: Java return keyword is used to return from a method when its execution is complete.
35. short: Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. static: Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
37. strictfp: Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. super: Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
39. switch: The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. synchronized: Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. this: Java this keyword can be used to refer the current object in a method or constructor.
42. throw: The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.
43. throws: The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.
44. transient: Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. try: Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. void: Java void keyword is used to specify that a method does not have a return value.
47. volatile: Java volatile keyword is used to indicate that a variable may change asynchronously.
48. while: Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use the while loop.

LITERAL

In Java, a literal is a fixed value that is directly written into the code and represents the value of a variable. Literals are used to assign values to variables or constants, and they are of various data types, such as numbers, characters, strings, and booleans.

TYPES OF LITERALS IN JAVA:

1. **Integer Literals:**

Represent whole numbers.

Examples:

- Decimal: 10, 123
- Binary (prefix 0b or 0B): 0b1010 (equals 10)
- Octal (prefix 0): 012 (equals 10)
- Hexadecimal (prefix 0x or 0X): 0xA (equals 10)

2. **Floating-Point Literals:**

Represent decimal or real numbers.

Examples:

- 3.14, 2.0, 1.5e3 (equals $1.5 \times 10^3 = 1500$)
- By default, floating-point literals are of type double. Use f or F for float, e.g., 3.14f.

3. **Character Literals:**

Represent a single character enclosed in single quotes (').

Examples:

- 'A', 'z', '1', '\n' (newline), '\u0061' (Unicode character for 'a')

4. **String Literals:**

Represent a sequence of characters enclosed in double quotes (").

Examples:

- "Hello, World!", "Java", "123"

5. **Boolean Literals:**

Represent true or false values.

Examples:

- true, false

6. **Null Literal:**

Represents the null reference. It is used to indicate that an object reference does not point to any memory location.

Example:

- null

7. **Underscore in Literals:**

Since Java 7, underscores (_) can be used in numeric literals for better readability.

Examples:

- 1_000_000 (equals 1000000)
- 0b1010_1010 (binary literal with underscores)

Example in Code:

```
public class LiteralsExample {
    public static void main(String[] args) {
        int decimal = 10;    // Integer literal
        int binary = 0b1010; // Binary literal
        int octal = 012;     // Octal literal
        int hex = 0xA;       // Hexadecimal literal

        float pi = 3.14f;    // Floating-point literal
        double largeNum = 1.5e3; // Double literal

        char letter = 'A';   // Character literal
        String message = "Hello, Java!"; // String literal

        boolean isJavaFun = true; // Boolean literal

        Object obj = null;    // Null literal

        System.out.println("Decimal: " + decimal);
        System.out.println("Binary: " + binary);
        System.out.println("Octal: " + octal);
        System.out.println("Hexadecimal: " + hex);
        System.out.println("Pi: " + pi);
        System.out.println("Large Number: " + largeNum);
        System.out.println("Letter: " + letter);
        System.out.println("Message: " + message);
        System.out.println("Is Java Fun? " + isJavaFun);
        System.out.println("Object: " + obj);
    }
}
```

IDENTIFIER,

In Java (and most programming languages), an **identifier** is the name used to identify variables, methods, classes, or other user-defined elements in a program. It is a sequence of characters that adheres to specific naming rules.

KEY CHARACTERISTICS OF IDENTIFIERS:

1. User-Defined Names:

Identifiers are names given by the programmer to elements like variables, methods, classes, etc.

2. Rules for Identifiers:

- **Allowed Characters:** Identifiers can include letters (A-Z, a-z), digits (0-9), underscore (`_`), and dollar sign (`$`).
- **Cannot Start with a Digit:** Identifiers must begin with a letter, an underscore, or a dollar sign but not a digit.
 -  `myVar`
 -  `1variable`
- **No Reserved Words:** Identifiers cannot be the same as Java's reserved keywords (e.g., `class`, `int`, `public`).
 -  `myClass`
 -  `class`

- **Case-Sensitive:** Identifiers are case-sensitive. For example, MyVariable and myvariable are different identifiers.
 - **Unlimited Length:** Java identifiers can be of any length.
3. **Convention (Not Mandatory):**
While not enforced, these naming conventions improve code readability:
- Use **camelCase** for variables and methods (myVariable, calculateSum).
 - Use **PascalCase** for class names (MyClass, UserProfile).
 - Use **UPPER_CASE_WITH_UNDERSCORES** for constants (MAX_VALUE, PI).
 - Avoid using \$ unless auto-generated (e.g., by a tool).

EXAMPLES OF VALID AND INVALID IDENTIFIERS:

VALID IDENTIFIERS:

- myVariable
- _temp
- \$price
- employeeCount
- MAX_VALUE

INVALID IDENTIFIERS:

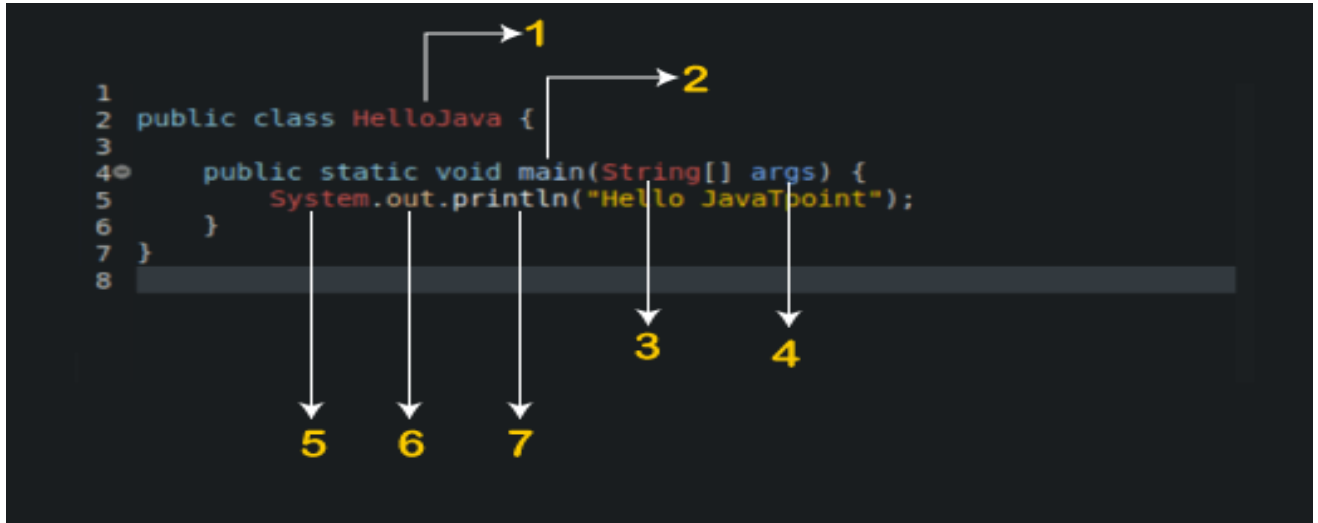
- 1stValue (starts with a digit)
- #price (contains an invalid character #)
- int (reserved keyword)
- employee-name (contains a hyphen -)

Example in Code:

```
public class IdentifierExample {
    public static void main(String[] args) {
        int age = 25;           // 'age' is an identifier for the variable
        double _price = 99.99;  // '_price' is a valid identifier
        String $name = "John";  // '$name' is a valid identifier
        final int MAX_LIMIT = 100; // 'MAX_LIMIT' is an identifier for a constant

        System.out.println("Age: " + age);
        System.out.println("Price: " + _price);
        System.out.println("Name: " + $name);
        System.out.println("Max Limit: " + MAX_LIMIT);
    }
}
```

In the above example, we have used the following identifiers



- HelloJava (Class name)
- main (main method)
- String (Predefined Class name)
- args (String variables)
- System (Predefined class)
- out (Variable name)
- println (method)

Rules for Java Identifiers

There are some rules and conventions for declaring the identifiers in Java. If the identifiers are not properly declared, we may get a compile-time error. Following are some rules and conventions for declaring identifiers:

- A valid identifier must have characters [A-Z] or [a-z] or numbers [0-9], and underscore (_) or a dollar sign (\$). For example, @java is not a valid identifier because it contains a special character @.
- There should not be any space in an identifier. For example, java language is an invalid identifier.
- An identifier should not contain a number at the starting. For example, 123java is an invalid identifier.
- An identifier should be of length 4-15 letters only. However, there is no limit on its length. But, it is good to follow the standard conventions.
- We cannot use the Java reserved keywords as an identifier such as int, float, double, char, etc. For example, int double is an invalid identifier in Java.
- An identifier should not be any query language keywords such as SELECT, FROM, COUNT, DELETE, etc.

Whitespace in Java refers to any character or sequence of characters that represent horizontal or vertical space in the code. It is used to improve code readability and formatting but does not affect the execution of the program.

TYPES OF WHITESPACE:

1. **Spaces:** Blank spaces ().
2. **Tabs:** Indentation spaces (usually created using the Tab key).
3. **Line Breaks:** Newline characters (\n or \r).

ROLE OF WHITESPACE:

1. **Code Separation:** Used to separate tokens in code such as keywords, identifiers, and literals.

- Example:

```
int a = 10; // Whitespace separates `int`, `a`, `=`, and `10`.
```

Improving Readability: Makes the code more organized and easier to read.

- Example:

```
if (x > 0) {  
    System.out.println("Positive");  
}
```

versus

```
if(x>0){System.out.println("Positive");}
```

Ignored by Compiler: The compiler ignores extra whitespace unless it's within string literals.

OPERATORS (ARITHMETIC, RELATIONAL, BOOLEAN LOGICAL, BITWISE LOGICAL, ASSIGNMENT, UNARY, SHIFT, SPECIAL OPERATORS)

Operator Type	Operator	Description	Example
Arithmetic	+	Addition	a + b
	-	Subtraction	a - b
	*	Multiplication	a * b
	/	Division	a / b
	%	Modulus (remainder)	a % b
Relational	==	Equal to	a == b
	!=	Not equal to	a != b
	<	Less than	a < b
	>	Greater than	a > b
	<=	Less than or equal to	a <= b
	>=	Greater than or equal to	a >= b
Boolean Logical	&&	Logical AND	(a > b) && (b > c)
		Logical OR	(a > b) (b > c)
	!	Logical NOT	!(a > b)
Bitwise Logical	&	Bitwise AND	a & b
		Bitwise OR	a b
	^	Bitwise XOR	a ^ b
	~	Bitwise Complement	~a
Assignment	=	Assign value	a = b
	+=	Add and assign	a += b
	-=	Subtract and assign	a -= b
	*=	Multiply and assign	a *= b
	/=	Divide and assign	a /= b
	%=	Modulus and assign	a %= b
Unary	+	Unary plus	+a
	-	Unary minus	-a
	++	Increment	++a, a++
	--	Decrement	--a, a--
	!	Logical NOT	!a
Shift	<<	Left shift	a << 2
	>>	Right shift (sign-preserving)	a >> 2
	>>>	Unsigned right shift	a >>> 2
Special Operators	?:	Ternary operator	a > b ? x : y
	instanceof	Tests if an object is an instance of a class/interface	obj instanceof String

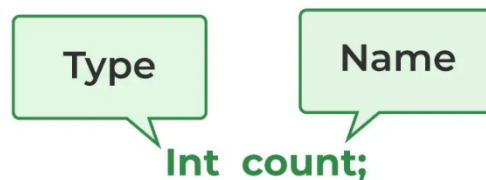
JAVA VARIABLE AND SCOPE OF VARIABLES

Variables are the containers for storing the data values or you can also call it a memory location name for the data. Every variable has a:

- Data Type – The kind of data that it can hold. For example, int, string, float, char, etc.
- Variable Name – To identify the variable uniquely within the scope.
- Value – The data assigned to the variable.

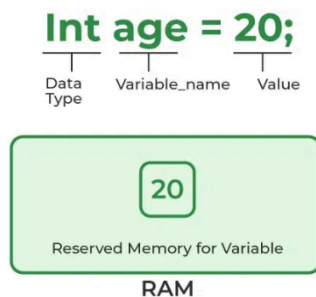
There are three types of variables in Java – Local, Instance, and Static.

How to Declare Java Variables?



- datatype: Type of data that can be stored in this variable.
- data_name: Name was given to the variable.

How to Initialize Java Variables?



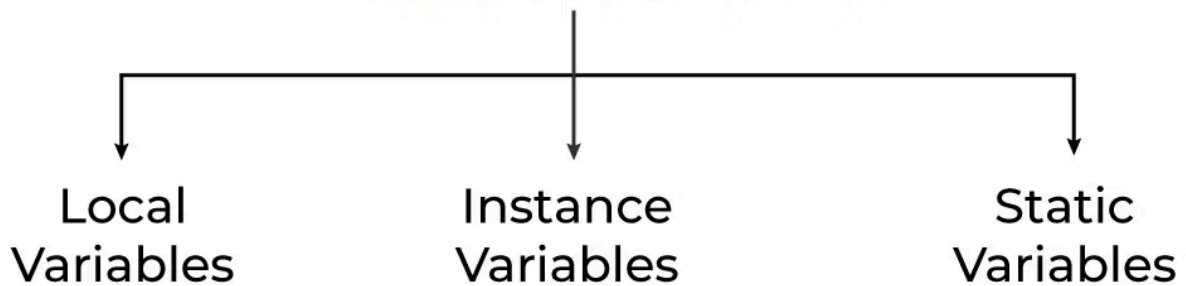
Types of Java Variables

Now let us discuss different types of variables which are listed as follows:

- Local Variables

- Instance Variables
- Static Variables

Type of Variable



Local Variables

A variable defined within a block or method or constructor is called a local variable.

The Local variable is created at the time of declaration and destroyed after exiting from the block or when the call returns from the function.

The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.

Initialization of the local variable is mandatory before using it in the defined scope.

// Java Program to show the use of local variables

```
class hello {  
    public static void main(String[] args)  
    {  
        // Declared a Local Variable  
        int var = 10;  
  
        // This variable is local to this main method only  
        System.out.println("Local Variable: " + var);  
    }  
}
```

INSTANCE VARIABLES

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.

Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.

Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.

Instance variables can be accessed only by creating objects.

```
// Java Program to show the use of
// Instance Variables
class hello {

    // Declared Instance Variable
    public String geek;
    public int i;
    public Integer I;
    public hello()
    {
        // Default Constructor
        // initializing Instance Variable
        this.geek = "Shubham Jain";
    }

    // Main Method
    public static void main(String[] args)
    {
        // Object Creation
        hello name = new hello();

        // Displaying O/P
        System.out.println("Geek name is: " + name.geek);
        System.out.println("Default value for int is " + name.i);

        // toString() called internally
        System.out.println("Default value for Integer is " + name.I);
    }
}
```

STATIC VARIABLES

Static variables are also known as class variables.

These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.

Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.

Static variables are created at the start of program execution and destroyed automatically when execution ends.

Initialization of a static variable is not mandatory. Its default value is dependent on

If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won't halt the program. The compiler will replace the object name with the class name automatically.

If we access a static variable without the class name, the compiler will automatically append the class name. But for accessing the static variable of a different class, we must mention the class name as 2 different classes might have a static variable with the same name.

Static variables cannot be declared locally inside an instance method.

```
// Java Program to show the use of
// Static variables

class hello {
    // Declared static variable
    public static String geek = "Shubham Jain";

    public static void main(String[] args)
    {

        // geek variable can be accessed without object
        // creation Displaying O/P GFG.geek --> using the
        // static variable
        System.out.println("Geek Name is : " + hello.geek);

        // static int c = 0;
        // above line, when uncommented,
        // will throw an error as static variables cannot be
        // declared locally.
    }
}
```

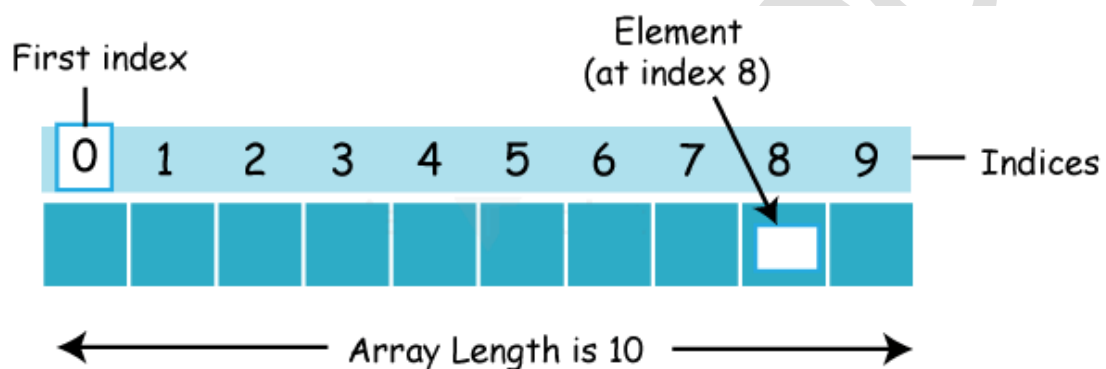
JAVA ARRAYS

An array is typically a grouping of elements of the same kind that are stored in a single, contiguous block of memory.

Java array is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

An object of a dynamically formed class is called an array in Java. Java arrays implement the Serializable and Cloneable interfaces and derive from the Object class. In Java, an array can hold objects or primitive values. Java allows us to create single- or multi-dimensional arrays, just like C/C++ does.



Advantages

- Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.
- Random access: We can get any data located at an index position.

Disadvantages

- Size Limit: Arrays have a fixed size and do not grow dynamically at runtime.

Types of Arrays in java

- One-dimensional
- Rectangular
- Jagged

In Java, a one-dimensional array is a data structure that allows you to store multiple values of the same data type in a single variable, accessed using an index. Here's a guide to declaring, initializing, and using a one-dimensional array:

Declaration and Initialization

```
dataType[] arrayName;
```

Instantiation:

arrayName = new dataType[size];
Combined Declaration and Instantiation:
dataType[] arrayName = new dataType[size];

Initialization with Values:
dataType[] arrayName = {value1, value2, value3, ...};

Example:

```
public class ODA {  
    public static void main(String[] args) {  
        // Declare and create an array of integers  
        int[] numbers = new int[5];  
  
        // Initialize the array  
        numbers[0] = 10;  
        numbers[1] = 20;  
        numbers[2] = 30;  
        numbers[3] = 40;  
        numbers[4] = 50;  
  
        // Access and print array elements using a loop  
        System.out.println("Array elements:");  
        for (int i = 0; i < numbers.length; i++) {  
            System.out.println("Element at index " + i + ": " + numbers[i]);  
        }  
  
        // Declare, initialize, and print in one step  
        String[] fruits = {"Apple", "Banana", "Cherry", "Date"};  
        System.out.println("\nFruits array:");  
        for (String fruit : fruits) {  
            System.out.println(fruit);  
        }  
    }  
}
```

RECTANGULAR ARRAY

A rectangular array in Java is essentially a two-dimensional array where each row has the same number of columns. This is also known as a matrix or a grid.

Declaration and Initialization
dataType[][] arrayName;

Instantiation
arrayName = new dataType[rows][columns];

Combined Declaration and Instantiation:
dataType[][] arrayName = new dataType[rows][columns];

Initialization with Values:
dataType[][] arrayName = {
 {value1, value2, value3},
 {value4, value5, value6},
 {value7, value8, value9}
};

Example Code

```
public class RectangularArrayExample {
    public static void main(String[] args) {
        // Declare and initialize a 2D array with specified dimensions
        int[][] matrix = new int[3][3];

        // Populate the array
        int value = 1;
        for (int i = 0; i < matrix.length; i++) { // Loop over rows
            for (int j = 0; j < matrix[i].length; j++) { // Loop over columns
                matrix[i][j] = value++;
            }
        }

        // Print the array
        System.out.println("Rectangular Array (Matrix):");
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }

        // Another way to declare and initialize
        int[][] preDefinedMatrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        System.out.println("\nPredefined Rectangular Array:");
        for (int i = 0; i < preDefinedMatrix.length; i++) {
            for (int j = 0; j < preDefinedMatrix[i].length; j++) {
                System.out.print(preDefinedMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

JAGGED ARRAY IN JAVA

A jagged array is an array of arrays in which each row (sub-array) can have a different number of columns. It is non-rectangular and provides flexibility for handling data structures where rows have varying lengths.

In Java:

Each row of a jagged array is a separate array that can be initialized independently. It is useful when you need to represent irregular structures, such as a triangle of numbers or other datasets with variable-length rows.

A jagged array is a collection of arrays where each sub-array can have a different size. It is declared and accessed using the same syntax as a two-dimensional array but provides the flexibility of varying row sizes.

Declaration and Initialization of Jagged array

Syntax:

```
datatype[][] arrayName = new datatype[numRows][];
```

```

arrayName[0] = new datatype[numColumns1];
arrayName[1] = new datatype[numColumns2];
...
arrayName[numRows-1] = new datatype[numColumnsN];

```

Example:

```

public class JAE {
    public static void main(String[] args) {
        // Declare a jagged array with 3 rows
        int[][] jaggedArray = new int[3][];

        // Initialize each row with a different size
        jaggedArray[0] = new int[2]; // First row has 2 columns
        jaggedArray[1] = new int[3]; // Second row has 3 columns
        jaggedArray[2] = new int[1]; // Third row has 1 column

        // Populate the jagged array
        int value = 1;
        for (int i = 0; i < jaggedArray.length; i++) {
            for (int j = 0; j < jaggedArray[i].length; j++) {
                jaggedArray[i][j] = value++;
            }
        }

        // Print the jagged array
        System.out.println("Jagged Array:");
        for (int i = 0; i < jaggedArray.length; i++) {
            for (int j = 0; j < jaggedArray[i].length; j++) {
                System.out.print(jaggedArray[i][j] + " ");
            }
            System.out.println();
        }
    }
}

```

WHAT IS COMMAND-LINE ARGUMENTS IN JAVA?

Command-line arguments are inputs passed to a Java program during execution through the command line or terminal. These arguments are provided as a sequence of strings and can be used to customize the behavior of the program without modifying the code.

How Command-Line Arguments Work

The main method in Java accepts an array of strings (String[] args) as a parameter.

These strings represent the arguments passed to the program.

Arguments are separated by spaces when passed via the terminal.

Syntax:

```
public static void main(String[] args) {  
    // args is an array of strings containing command-line arguments  
}
```

Basic Example

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

WRITING, COMPILING AND EXECUTING BASIC JAVA PROGRAM

Writing a Java Program

Code Example

Save the following code in a text file with the name HelloWorld.java:

// A simple Java program

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

Compiling the Program

Compilation Steps

Open a terminal or command prompt.

Navigate to the directory where the HelloWorld.java file is saved.

cd path/to/directory

Compile the program using the javac command:

javac HelloWorld.java

What Happens During Compilation?

The javac command compiles the HelloWorld.java file and generates a bytecode file named HelloWorld.class.

This bytecode file can be executed by the Java Virtual Machine (JVM).

Executing the Program

Execution Steps

Run the compiled bytecode using the java command:

```
java HelloWorld
```

Example Walkthrough

Code: Basic Addition Program

Save the following code as Addition.java:

```
public class Addition {  
    public static void main(String[] args) {  
        int num1 = 10;  
        int num2 = 20;  
        int sum = num1 + num2;  
        System.out.println("Sum: " + sum);  
    }  
}
```

DECISION STATEMENTS (IF, SWITCH)

Decision statements are used to make decisions based on certain conditions and execute a specific block of code. Java provides the following decision-making statements:

- if Statement
- if-else Statement
- if-else-if Ladder
- switch Statement

IF STATEMENT

The if statement executes a block of code if the specified condition is true.

Syntax

```
if (condition) {  
    // Code to execute if condition is true  
}
```

Example

```
public class IfExample {
```

```

public static void main(String[] args) {
    int number = 10;
    if (number > 5) {
        System.out.println("Number is greater than 5.");
    }
}

```

IF-ELSE STATEMENT

The if-else statement executes one block of code if the condition is true and another block if the condition is false.

Syntax:

```

if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}

```

Example

```

public class IfElseExample {
    public static void main(String[] args) {
        int number = 4;

        if (number % 2 == 0) {
            System.out.println("Number is even.");
        } else {
            System.out.println("Number is odd.");
        }
    }
}

```

IF-ELSE-IF LADDER

The if-else-if ladder checks multiple conditions in sequence. The first condition that evaluates to true will execute its corresponding block.

Syntax:

```

if (condition1) {
    // Code for condition1
} else if (condition2) {
    // Code for condition2
} else if (condition3) {
    // Code for condition3
} else {
    // Code if none of the conditions are true
}

```

Example

```

public class IfElseIfExample {

```

```

public static void main(String[] args) {
    int marks = 85;

    if (marks >= 90) {
        System.out.println("Grade: A+");
    } else if (marks >= 80) {
        System.out.println("Grade: A");
    } else if (marks >= 70) {
        System.out.println("Grade: B");
    } else {
        System.out.println("Grade: C");
    }
}
}

```

SWITCH STATEMENT

The switch statement allows the execution of one block of code from multiple options, based on the value of an expression.

Syntax

```

switch (expression) {
    case value1:
        // Code for value1
        break;
    case value2:
        // Code for value2
        break;
    ...
    default:
        // Code if no case matches
}

```

Example

```

public class SwitchExample {
    public static void main(String[] args) {
        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;
            case 4:
                System.out.println("Thursday");
                break;
            case 5:
                System.out.println("Friday");
                break;
            default:
                System.out.println("Weekend");
        }
    }
}

```

- The if family of statements is suitable for conditions with ranges or complex logical expressions.
- The switch statement is preferable for equality checks on a single variable, especially with multiple discrete values.

- Use break in switch to prevent fall-through behavior (i.e., executing subsequent cases unintentionally).

LOOPING STATEMENTS (FOR, WHILE, DO-WHILE)

Loops in Java are used to execute a block of code repeatedly as long as a specified condition is true. Java provides three types of looping statements:

- for Loop
- while Loop
- do-while Loop

FOR LOOP

The for loop is used when the number of iterations is known beforehand.

Syntax

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

Example: Print Numbers from 1 to 5

```
public class ForLoopExample {  
    public static void main(String[] args) {  
        for (int i = 1; i <= 5; i++) {  
            System.out.println("Number: " + i);  
        }  
    }  
}
```

WHILE LOOP

The while loop is used when the number of iterations is not known, and it depends on a condition.

Syntax

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

Example: Print Numbers from 1 to 5

```
public class WhileLoopExample {  
    public static void main(String[] args) {  
        int i = 1;
```

```

        while (i <= 5) {
            System.out.println("Number: " + i);
            i++;
        }
    }
}

```

DO-WHILE LOOP

The do-while loop is similar to the while loop, but it executes the block of code at least once, even if the condition is false.

Syntax

```

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

```

Example: Print Numbers from 1 to 5

```

public class DowhileLoopExample {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println("Number: " + i);
            i++;
        } while (i <= 5);
    }
}

```

COMPARISON OF LOOPS

Feature	for Loop	while Loop	do-while Loop
When to Use	When iterations are known beforehand.	When iterations depend on a condition.	When the loop must execute at least once.
Condition Check	At the beginning of each iteration.	At the beginning of each iteration.	At the end of each iteration.
Execution Guarantee	Executes only if the condition is true.	Executes only if the condition is true.	Executes at least once.

Examples: Common Use Cases

Print Even Numbers from 1 to 10 (for Loop)

```

public class EvenNumbers {
    public static void main(String[] args) {
        for (int i = 2; i <= 10; i += 2) {
            System.out.println(i);
        }
    }
}

```

Find the Sum of Numbers from 1 to 5 (while Loop)

```

public class SumNumbers {
    public static void main(String[] args) {
        int sum = 0, i = 1;
        while (i <= 5) {
            sum += i;
            i++;
        }
        System.out.println("Sum: " + sum);
    }
}

```

Display Menu until User Exits (do-while Loop)

```

import java.util.Scanner;
public class MenuExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int choice;

        do {
            System.out.println("1. Option 1");
            System.out.println("2. Option 2");
            System.out.println("3. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();
        } while (choice != 3);

        System.out.println("Goodbye!");
    }
}

```

STATIC AND NON-STATIC MEMBERS

Static Members:

Static members belong to the class rather than to any specific instance of the class. This means they are shared among all instances (objects) of the class. Static members can be accessed without creating an instance of the class.

Characteristics:

- Declaration: A static member is declared using the static keyword.
- Access: Static members can be accessed directly using the class name, without creating an object.
- Lifetime: Static members exist as long as the class is loaded into memory (i.e., the entire program's runtime).
- Memory Allocation: Only one copy of a static member is shared across all instances of the class.
- Usage: Typically used for constants, utility methods, or fields that are common to all instances.

Example:

```
class MyClass {
    static int counter = 0; // static variable

    static void increment() { // static method
        counter++;
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass.increment(); // Accessing static method without creating an object
        System.out.println(MyClass.counter); // Output will be 1
    }
}
```

Non-Static Members:

Non-static members belong to an instance of the class. Each object (instance) of the class has its own copy of these members.

Characteristics:

- Declaration: A non-static member is declared without the static keyword.
- Access: Non-static members must be accessed through an instance of the class.
- Lifetime: Non-static members exist as long as the object is alive (i.e., until the object is destroyed).
- Memory Allocation: Each object of the class gets its own copy of non-static members.
- Usage: Typically used when the value of the member is specific to each instance of the class.

Example:

```

class MyClass {
    int counter = 0; // non-static variable

    void increment() { // non-static method
        counter++;
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();

        obj1.increment();
        obj2.increment();

        System.out.println(obj1.counter); // Output will be 1
        System.out.println(obj2.counter); // Output will be 1
    }
}

```

KEY DIFFERENCES:

Aspect	Static Members	Non-Static Members
Association	Belong to the class.	Belong to instances of the class.
Access	Can be accessed using the class name.	Must be accessed through an object.
Memory	Only one copy for the entire class.	Each object has its own copy.
Lifetime	Exists as long as the class is loaded.	Exists as long as the object exists.
Usage	Used for shared or common behavior.	Used for object-specific behavior.

CONCLUSION:

- **Static members** are useful for behavior that should be shared across all instances, such as counters or constants.
- **Non-static members** are useful for data that is unique to each object, like instance variables that hold different values for different objects.

A constructor in Java is a special block of code used to initialize objects. It has the same name as the class and is called automatically when an object of the class is created. Constructors are essential for setting up the initial state of an object.

Characteristics of Constructors

- Same Name as Class: A constructor must have the same name as the class.
- No Return Type: Constructors do not specify a return type, not even void.
- Automatic Invocation: Called when an object is created using the new keyword.
- Purpose: Primarily used to initialize variables of a class or perform setup tasks.

Types of Constructors

- Default Constructor (No-Argument Constructor)
- Parameterized Constructor

Default Constructor

A default constructor is a constructor that does not take any parameters. If no constructor is defined in a class, Java provides a default constructor automatically. However, if you define a constructor, the default constructor is no longer provided automatically.

CHARACTERISTICS:

- Has no parameters.
- Initializes fields with default values:
 - 0 for numeric types.
 - null for objects.
 - false for boolean.
- Useful for creating objects without any specific initial values.

Example

```
class MyClass {
    int num; // Default value will be 0
    String text; // Default value will be null

    // Default constructor
    MyClass() {
        System.out.println("Default constructor called");
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass(); // Default constructor is called
        System.out.println("num: " + obj.num); // Output: num: 0
        System.out.println("text: " + obj.text); // Output: text: null
    }
}
```

PARAMETERIZED CONSTRUCTOR

A parameterized constructor takes arguments to initialize an object with specific values. It provides more flexibility by allowing different initial values for different objects.

CHARACTERISTICS:

- Takes parameters to set the values of fields.
- Enables object initialization with user-defined values.

Example

```
class MyClass {
    int num;
    String text;

    // Parameterized constructor
    MyClass(int number, String str) {
        num = number;
        text = str;
        System.out.println("Parameterized constructor called");
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass(42, "Hello");
        MyClass obj2 = new MyClass(100, "World");

        System.out.println("obj1 - num: " + obj1.num + ", text: " + obj1.text);
        // Output: obj1 - num: 42, text: Hello
        System.out.println("obj2 - num: " + obj2.num + ", text: " + obj2.text);
        // Output: obj2 - num: 100, text: World
    }
}
```

KEY DIFFERENCES BETWEEN DEFAULT AND PARAMETERIZED CONSTRUCTOR

Aspect	Default Constructor	Parameterized Constructor
Parameters	No parameters.	Takes one or more parameters.
Initialization	Initializes fields with default values.	Initializes fields with user-defined values.
Purpose	Creates generic objects.	Creates specific objects with custom values.
Provided by Java	Automatically provided if no constructor exists.	Must be explicitly defined by the programmer.

KEY POINTS

- If you define any constructor (including a parameterized one), Java does not provide a default constructor automatically.
- Constructors can be overloaded, allowing both default and parameterized versions in the same class.
- The `this` keyword can be used to differentiate between instance variables and parameters.

CONCLUSION

- Use the **default constructor** when you need to create objects with default or generic values.

- Use the **parameterized constructor** to initialize objects with specific values during their creation.

FINALIZE () METHOD IN JAVA

The finalize () method in Java is a method of the Object class used for cleanup operations before an object is removed from memory by the garbage collector. It provides an opportunity for an object to perform any necessary cleanup (like releasing resources) before being destroyed.

KEY POINTS ABOUT FINALIZE()

1. Defined in the Object Class:

- Every Java class inherits the finalize() method from the Object class.
- Its default implementation does nothing.

2. Purpose:

- Typically used to release non-Java resources like file handles, database connections, or native code resources.
- Not recommended for general cleanup, as it can introduce unpredictable behavior.

3. Syntax:

```
protected void finalize() throws Throwable {  
  
    // Cleanup code here  
  
}
```

1. Called by the Garbage Collector:

- The garbage collector calls the finalize() method before reclaiming memory for the object.
- There is no guarantee when or even if finalize() will be called, as it depends on the garbage collector.

Example of finalize() Usage

```
class MyClass {
    // Constructor
    MyClass() {
        System.out.println("Object created.");
    }

    // Overriding finalize method
    @Override
    protected void finalize() throws Throwable {
        System.out.println("finalize() method called.");
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass();
        obj1 = null; // Eligible for garbage collection

        // Requesting garbage collection
        System.gc();

        System.out.println("End of program.");
    }
}
```

IMPORTANT CONSIDERATIONS

1. Unpredictability:

- The finalize() method is not guaranteed to be called immediately when an object is no longer referenced.
- It may never be called if the program ends before garbage collection.

2. Deprecated in Java 9 and Later:

- The finalize() method was deprecated starting with Java 9 due to its unpredictability and potential for misuse.
- Alternatives like try-with-resources or explicitly closing resources (close()) are preferred.

3. Performance Impact:

- Relying on finalize() can lead to performance issues, as objects requiring finalization take longer to be garbage collected.

CONCLUSION

While the finalize() method was historically used for resource cleanup, it is now considered outdated and unreliable. Use modern techniques like try-with-resources or explicit cleanup methods for better performance, control, and predictability.

Constructor overloading in Java refers to defining multiple constructors in the same class with different parameter lists. This allows objects of the class to be created in different ways, providing flexibility for initializing variables.

KEY FEATURES OF CONSTRUCTOR OVERLOADING

1. Multiple Constructors:

- A class can have multiple constructors with the same name but different parameter types, numbers, or orders.

2. Compile-Time Polymorphism:

- Constructor overloading is an example of compile-time polymorphism, as the compiler determines which constructor to invoke based on the arguments provided.

3. Flexibility:

- Overloading constructors provide various ways to create objects with default, partial, or full initialization.
-

RULES FOR CONSTRUCTOR OVERLOADING

1. The constructors must have the same name as the class.
 2. The parameter lists must differ in at least one of the following:
 - Number of parameters.
 - Types of parameters.
 - Order of parameters.
-

EXAMPLE OF CONSTRUCTOR OVERLOADING

```
class Student {
    String name;
    int age;
    String course;

    // Constructor 1: No parameters
    Student() {
        name = "Unknown";
        age = 0;
        course = "Not Assigned";
    }

    // Constructor 2: One parameter
    Student(String studentName) {
        name = studentName;
        age = 0;
        course = "Not Assigned";
    }

    // Constructor 3: Two parameters
    Student(String studentName, int studentAge) {
        name = studentName;
        age = studentAge;
        course = "Not Assigned";
    }
}
```

```

// Constructor 4: Three parameters
Student(String studentName, int studentAge, String studentCourse) {
    name = studentName;
    age = studentAge;
    course = studentCourse;
}

void display() {
    System.out.println("Name: " + name + ", Age: " + age + ", Course: " + course);
}

}

public class Test {
    public static void main(String[] args) {
        // Using different constructors
        Student student1 = new Student();
        Student student2 = new Student("Alice");
        Student student3 = new Student("Bob", 20);
        Student student4 = new Student("Charlie", 22, "Computer Science");

        // Displaying student information
        student1.display();
        student2.display();
        student3.display();
        student4.display();
    }
}

```

ADVANTAGES OF CONSTRUCTOR OVERLOADING

1. **Improved Readability:** Allows creating objects with varying initialization needs in a clear manner.
2. **Code Reusability:** Reuses the same constructor name, reducing code redundancy.
3. **Ease of Use:** Simplifies object creation by offering different ways to initialize objects.

Method overloading in Java is a feature that allows a class to have multiple methods with the same name but different parameters. It enhances code readability and usability by providing flexibility to call a method with varying arguments.

KEY FEATURES OF METHOD OVERLOADING:

1. **Same Method Name:** All overloaded methods must share the same name.
2. **Different Parameter Lists:** Methods can differ in:
 - The number of parameters.
 - The type of parameters.
 - The order of parameters.
3. **Compile-time Polymorphism:** Method overloading is resolved during compile-time.

RULES FOR METHOD OVERLOADING:

- Methods can have different numbers of parameters.
- Methods can have the same number but different types of parameters.
- Changing the return type **alone** does not constitute overloading.

EXAMPLES OF METHOD OVERLOADING:

```
public class OverloadingExample {
    // Method with no parameters
    public void display() {
        System.out.println("Display method with no arguments.");
    }

    // Method with one parameter
    public void display(int a) {
        System.out.println("Display method with one argument: " + a);
    }

    // Method with two parameters
    public void display(int a, String b) {
        System.out.println("Display method with two arguments: " + a + ", " + b);
    }

    // Method with different parameter order
    public void display(String b, int a) {
        System.out.println("Display method with reversed arguments: " + b + ", " + a);
    }

    public static void main(String[] args) {
        OverloadingExample obj = new OverloadingExample();

        obj.display();
        obj.display(10);
        obj.display(20, "Hello");
        obj.display("World", 30);
    }
}
```

BENEFITS OF METHOD OVERLOADING:

1. Improves code clarity by using the same method name for related operations.
2. Reduces redundancy, making the code more maintainable.
3. Supports cleaner and organized APIs for libraries and frameworks.

@bcbcaschool

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that we can create new classes that are built upon existing classes. When we inherit methods from an existing class, we can reuse methods and fields of the parent class. However, we can add new methods and fields in your current class also.

WHAT IS INHERITANCE?

Inheritance in Java enables a class to inherit properties and actions from another class, called a superclass or parent class. A class derived from a superclass is called a subclass or child group. Through inheritance, a subclass can access members of its superclass (fields and methods), enforce reuse rules, and encourage hierarchy.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

The IS-A relationship in Java refers to the relationship between a subclass (child class) and its superclass (parent class) when inheritance is used. It indicates that the subclass is a specialized form of the superclass. Essentially, IS-A means "is a type of," which is a way of expressing the relationship in real-world or object-oriented terms.

Example in Real Life:

- A Dog IS-A Animal.
- A Car IS-A Vehicle.
- A Manager IS-A Employee.

Why use inheritance in Java?

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance: -

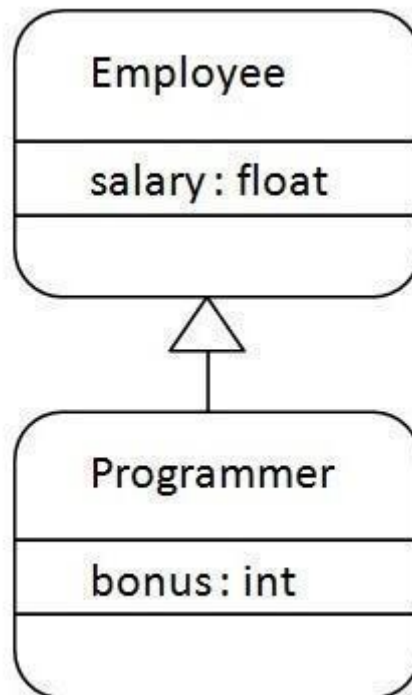
```
class Subclass-name extends Superclass-name
```

```
{  
    //methods and fields  
}
```

The `extends` keyword indicates that we are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class that is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, **Programmer** is the subclass and **Employee** is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that **Programmer** is a type of **Employee**.

```
class Employee{  
    float salary=40000;  
}
```

```

class Programmer extends Employee{
    int bonus=10000;
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}

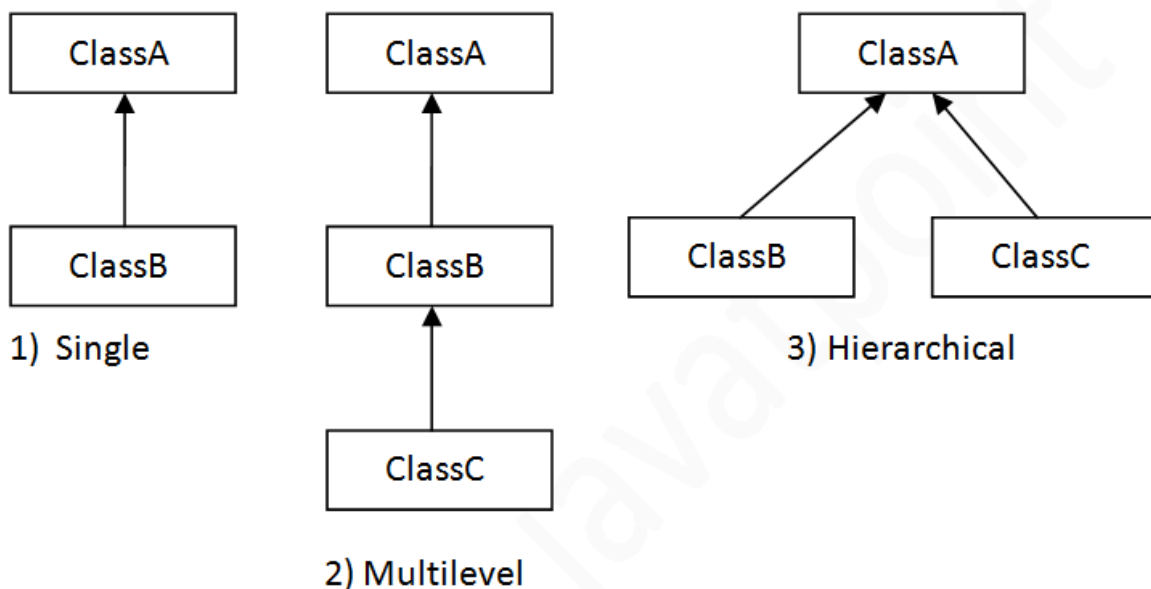
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

TYPES OF INHERITANCE IN JAVA

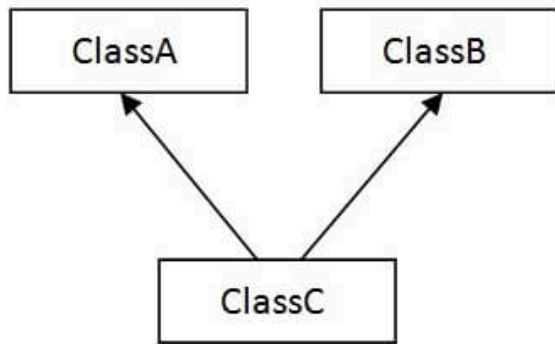
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

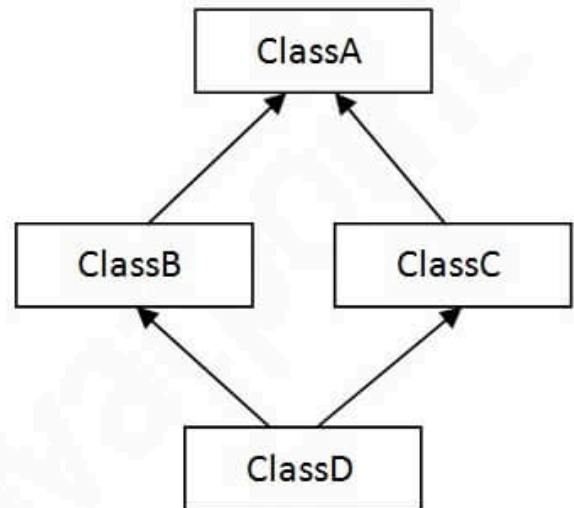


Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

SINGLE INHERITANCE EXAMPLE

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```

class Animal
{
    void eat()
    {
        System.out.println("eating...");
    }
}
class Dog extends Animal
{
    void bark()
    {
        System.out.println("barking...");
    }
}
class TestInheritance
{
    public static void main(String args[]){
        Dog d=new Dog();
        d.bark();
        d.eat();
    }
}
  
```

Output:

barking...

eating...

MULTILEVEL INHERITANCE EXAMPLE

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal
{
    void eat()
    {
        System.out.println("eating...");
    }
}
class Dog extends Animal
{
    void bark()
    {
        System.out.println("barking...");
    }
}
class BabyDog extends Dog
{
    void weep()
    {
        System.out.println("weeping...");
    }
}
class TestInheritance2
{
    public static void main(String args[])
    {
        BabyDog d=new BabyDog();
        d.weep();
        d.bark();
        d.eat();
    }
}
```

Output:

weeping...

barking...

eating...

HIERARCHICAL INHERITANCE EXAMPLE

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```

class Animal
{
    void eat()
    {
        System.out.println("eating...");
    }
}
class Dog extends Animal
{
    void bark()
    {
        System.out.println("barking...");
    }
}
class Cat extends Animal
{
    void meow()
    {
        System.out.println("meowing...");
    }
}
class TestInheritance3
{
    public static void main(String args[])
    {
        Cat c=new Cat();
        c.meow();
        c.eat();
        //c.bark();//C.T.Error
    }
}

```

Why multiple inheritance is not supported in Java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Suppose there are three classes A, B, and C. The C class inherits A and B classes. If A and B classes have the same method and we call it from child class object, there will be ambiguity to call the method of A or B class.

```

class A
{
    void msg()
    {
        System.out.println("Hello");
    }
}
class B
{
    void msg()
    {
        System.out.println("Welcome");
    }
}
class C extends A,B
{//suppose if it were

    public static void main(String args[])
    {
        C obj=new C();
        obj.msg();//Now which msg() method would be invoked?
    }
}

```

ACCESS MODIFIERS IN JAVA

There are two types of modifiers in Java: access modifiers and non-access modifiers.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

- **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```

class A
{
    private int data=40;
    private void msg()
    {
        System.out.println("Hello java");
    }
}
public class Simple
{
    public static void main(String args[])
    {
        A obj=new A();
        System.out.println(obj.data);//Compile Time Error
        obj.msg();//Compile Time Error
    }
}

```

Default

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

//save by A.java

```

package pack;
class A{
    void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}

```

Protected

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifier.

```
//save by A.java
package pack;
public class A
{
    protected void msg()
    {
        System.out.println("Hello");
    }
}
```

```
//save by B.java
package mypack;
import pack.*;

class B extends A
{
    public static void main(String args[]){
        B obj = new B();
        obj.msg();
    }
}
```

Public

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

```
//save by A.java
package pack;
public class A
{
    public void msg()
    {
        System.out.println("Hello");
    }
}
```

```
//save by B.java
package mypack;
import pack.*;

class B
{
    public static void main(String args[])
    {
        A obj = new A();
        obj.msg();
    }
}
```

Java.Lang Package Classes:-

JAVA.LANG.MATH

The java.lang.Math class in Java provides a collection of methods for performing basic numeric operations such as exponentiation, logarithms, trigonometry, and rounding. It is part of the java.lang package, which is automatically imported in every Java program.

Key Features of java.lang.Math Class:

Constants:

Math.PI → Value of π (3.141592653589793)

Common Methods:

Method	Description	Example
Math.abs(x)	Returns the absolute value of x.	Math.abs(-5) → 5
Math.max(a, b)	Returns the maximum of a and b.	Math.max(10, 20) → 20
Math.min(a, b)	Returns the minimum of a and b.	Math.min(10, 20) → 10
Math.sqrt(x)	Returns the square root of x.	Math.sqrt(25) → 5.0
Math.pow(a, b)	Returns a raised to the power of b.	Math.pow(2, 3) → 8.0
Math.round(x)	Rounds to the nearest integer.	Math.round(4.6) → 5
Math.ceil(x)	Rounds up to the nearest integer.	Math.ceil(4.3) → 5.0
Math.floor(x)	Rounds down to the nearest integer.	Math.floor(4.7) → 4.0
Math.random()	Returns a random number between 0.0 and 1.0.	Math.random() → 0.5432
Math.log(x)	Returns the natural logarithm of x.	Math.log(10)
Math.exp(x)	Returns e raised to the power of x.	Math.exp(2)

Trigonometric Methods:

Method	Description	Example
Math.sin(x)	Returns the sine of x (radians).	Math.sin(Math.PI / 2) → 1.0
Math.cos(x)	Returns the cosine of x (radians).	Math.cos(0) → 1.0
Math.tan(x)	Returns the tangent of x (radians).	Math.tan(Math.PI / 4) → 1.0
Math.toRadians(x)	Converts degrees to radians.	Math.toRadians(180) → π
Math.toDegrees(x)	Converts radians to degrees.	Math.toDegrees(Math.PI) → 180.0

Important Notes:

Most methods in Math return results in double type.

The class cannot be instantiated; it only provides static methods.

JAVA LANG WRAPPER CLASS

A Wrapper class in Java is one whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and, in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Use of Wrapper classes in Java:- Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Primitive Data Types and their Corresponding Wrapper Class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Autoboxing and Unboxing

The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double, etc.

Example:

```
// Java program to demonstrate Autoboxing
import java.util.ArrayList;
class Autoboxing {
    public static void main(String[] args)
    {
        char ch = 'a';

        // Autoboxing- primitive to Character object
        // conversion
        Character a = ch;

        ArrayList<Integer> arrayList
            = new ArrayList<Integer>();

        // Autoboxing because ArrayList stores only objects
        arrayList.add(25);

        // printing the values from object
        System.out.println(arrayList.get(0));
    }
}
```

Unboxing

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc

Example:

```
// Java program to demonstrate Unboxing
import java.util.ArrayList;
class Unboxing {
    public static void main(String[] args)
    {
        Character ch = 'a';

        // unboxing - Character object to primitive
        // conversion
        char a = ch;

        ArrayList<Integer> arrayList
            = new ArrayList<Integer>();
        arrayList.add(24);

        // unboxing because get method returns an Integer
        // object
        int num = arrayList.get(0);

        // printing the values from primitive data types
        System.out.println(num);
    }
}
```

Java Wrapper Classes Example:

```

//Java program to demonstrate Wrapping and UnWrapping
//in Classes
class wpclass {
    public static void main(String[] args)
    {
        // byte data type
        byte a = 1;

        // wrapping around Byte object
        Byte byteobj = new Byte(a);
        // Use with Java 9
        // Byte byteobj = Byte.valueOf(a);

        // int data type
        int b = 10;

        // wrapping around Integer object
        Integer intobj = new Integer(b);
        // Use with Java 9
        // Integer intobj = Integer.valueOf(b);

        // float data type
        float c = 18.6f;

        // wrapping around Float object
        Float floatobj = new Float(c);
        // Use with Java 9
        // Float floatobj = Float.valueOf(c);

        // double data type
        double d = 250.5;

        // Wrapping around Double object
        Double doubleobj = new Double(d);
        // Use with Java 9
        // Double doubleobj = Double.valueOf(d);

        // char data type
        char e = 'a';

        // wrapping around Character object
        Character charobj = e;

        // printing the values from objects
        System.out.println(
            "Values of Wrapper objects (printing as objects)");
        System.out.println("\nByte object byteobj: "
            + byteobj);
        System.out.println("\nInteger object intobj: "
            + intobj);
        System.out.println("\nFloat object floatobj: "
            + floatobj);
        System.out.println("\nDouble object doubleobj: "
            + doubleobj);
        System.out.println("\nCharacter object charobj: "
            + charobj);

        // objects to data types (retrieving data types from
        // objects) unwrapping objects to primitive data
        // types
        byte bv = byteobj;
        int iv = intobj;
        float fv = floatobj;
        double dv = doubleobj;
        char cv = charobj;

        // printing the values from data types
        System.out.println(

```

```

        "\nUnwrapped values (printing as data types)");
System.out.println("\nbyte value, bv: " + bv);
System.out.println("\nint value, iv: " + iv);
System.out.println("\nfloat value, fv: " + fv);
System.out.println("\ndouble value, dv: " + dv);
System.out.println("\nchar value, cv: " + cv);
    }
}

```

Example :

```

public class wpclass {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int i=15;
        Integer in=new Integer(i);
        System.out.println(in);
        // its called wrapping or boxing

        Integer in1 = i;
        System.out.println(in1);
        // its called autoboxing

        int x=in.intValue();
        System.out.println(x);
        // its called unboxing

        int y=in1;
        System.out.println(y);
        //its called autounboxing

        String str = "15061990";

        int num1= Integer.parseInt(str);
        System.out.println(num1);

    }

}

```

Description

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write -

```
String greeting = "Hello world!";
```

Creating String from Char Array Example

```
public class StringDemo {  
    public static void main(String args[]) {  
        char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };  
        String helloString = new String(helloArray);  
        System.out.println( helloString );  
    }  
}
```

String Length

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

The following program is an example of length(), method String class.

Getting Length of the String Example

```
public class StringDemo {  
    public static void main(String args[]) {  
        String palindrome = "Dot saw I was Tod";  
        int len = palindrome.length();  
        System.out.println( "String Length is : " + len );  
    }  
}
```

Concatenating Strings

```
public class StringDemo {  
    public static void main(String args[]) {  
        String string1 = "saw I was ";  
        System.out.println("Dot " + string1 + "Tod");  
    }  
}
```

String Methods

Method	Description	Example
length()	Returns the length of the string.	"Hello".length() → 5
charAt(index)	Returns the character at the specified index.	"Java".charAt(1) → 'a'
concat(str)	Concatenates the specified string.	"Java".concat(" World") → "Java World"
toLowerCase()	Converts all characters to lowercase.	"JAVA".toLowerCase() → "java"
toUpperCase()	Converts all characters to uppercase.	"java".toUpperCase() → "JAVA"
trim()	Removes whitespace from both ends.	" Java ".trim() → "Java"
substring(start, end)	Extracts part of the string.	"Hello".substring(1, 4) → "ello"
replace(old, new)	Replaces occurrences of a character/substring.	"abc".replace('a', 'x') → "xbc"
indexOf(char)	Returns the index of the first occurrence.	"Hello".indexOf('e') → 1
lastIndexOf(char)	Returns the last occurrence index.	"Hello".lastIndexOf('l') → 3
equals(str)	Compares two strings for equality.	"Java".equals("java") → false
equalsIgnoreCase(str)	Compares two strings, ignoring case.	"Java".equalsIgnoreCase("java") → true
split(regex)	Splits the string into an array.	"a,b,c".split(",") → ["a", "b", "c"]
startsWith(prefix)	Checks if the string starts with a given prefix.	"Java".startsWith("J") → true
endsWith(suffix)	Checks if the string ends with a given suffix.	"Java".endsWith("a") → true
isEmpty()	Checks if the string is empty.	"".isEmpty() → true

Example Program:

```

public class StringExample {
    public static void main(String[] args) {
        String str = " Hello, Java! ";

        System.out.println("Original String: '" + str + "'");
        System.out.println("Length: " + str.length());
        System.out.println("Trimmed: '" + str.trim() + "'");
        System.out.println("Uppercase: " + str.toUpperCase());
        System.out.println("Substring: " + str.substring(2, 7));
        System.out.println("Replace: " + str.replace("Java", "World"));
        System.out.println("Contains 'Java': " + str.contains("Java"));
    }
}

```

Example Showing Different References After Modification

```

public class StringImmutableProof {
    public static void main(String[] args) {
        String str1 = "Java";
        String str2 = str1; // Both str1 and str2 point to the same object

        str1 = str1 + " Programming"; // New string created, str1 now refers to new object

        System.out.println("str1: " + str1);
    }
}

```

```
System.out.println("str2: " + str2);

if (str1 == str2) {
    System.out.println("Both references are the same.");
} else {
    System.out.println("References are different, proving immutability.");
}
}
```

STRINGBUFFER CLASS IN JAVA

StringBuffer is a class in Java that represents a mutable sequence of characters. It provides an alternative to the immutable String class, allowing you to modify the contents of a string without creating a new object every time.

In Java, the StringBuffer class (part of the java.lang package) is used to create mutable (modifiable) strings. Unlike the String class, which creates immutable objects, StringBuffer allows changes to the string without creating new objects, making it more efficient for frequent string manipulations.

Key Features of StringBuffer:

- **Mutable:**
 - The contents of a StringBuffer object can be modified without creating a new object.
- **Thread-safe (Synchronized):**
 - It provides synchronization, making it suitable for multi-threaded environments.
 - Slightly slower than StringBuilder due to thread safety.
- **Efficient for frequent modifications:**
 - Ideal for scenarios involving repeated string modifications, such as appending or deleting characters.

Creating a StringBuffer Object

1. Using the default constructor (initial capacity 16 characters):

```
StringBuffer sb = new StringBuffer();
```

2. Specifying initial capacity:

```
StringBuffer sb = new StringBuffer(50);
```

3. Initializing with a string:

```
StringBuffer sb = new StringBuffer("Hello");
```

Common Methods of StringBuffer

Method	Description	Example
append(String str)	Adds a string to the end.	sb.append(" World") → "Hello World"
insert(int offset, String str)	Inserts a string at a specific index.	sb.insert(5, " Java") → "Hello Java"
replace(int start, int end, String str)	Replaces a part of the string.	sb.replace(0, 5, "Hi") → "Hi Java"
delete(int start, int end)	Deletes characters within a range.	sb.delete(0, 2) → "llo"
reverse()	Reverses the string.	"Java" → "avaJ"
length()	Returns the length of the string.	sb.length() → 5
capacity()	Returns the buffer capacity.	sb.capacity() → 16 (default)
charAt(int index)	Returns a character at the given index.	sb.charAt(0) → 'H'
setCharAt(int index, char ch)	Changes a character at the given index.	sb.setCharAt(0, 'h') → "hello"
substring(int start, int end)	Returns part of the string.	sb.substring(0, 5) → "Hello"

Example Usage of StringBuffer:

```

public class StringBufferExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");
        // Appending text
        sb.append(" World");
        System.out.println("After append: " + sb);
        // Inserting text
        sb.insert(5, " Java");
        System.out.println("After insert: " + sb);
        // Replacing text
        sb.replace(6, 10, "C++");
        System.out.println("After replace: " + sb);
        // Deleting text
        sb.delete(5, 9);
        System.out.println("After delete: " + sb);

        // Reversing string
        sb.reverse();
        System.out.println("After reverse: " + sb);
    }
}

```

Converting Between String and StringBuffer

```

StringBuffer to String:
StringBuffer sb = new StringBuffer("Hello");
String str = sb.toString();
System.out.println(str);

```

String to StringBuffer:

```
String str = "Hello";
StringBuffer sb = new StringBuffer(str);
System.out.println(sb);
```

Important Points to Remember

- StringBuffer is mutable, meaning its content can be modified without creating a new object.
- It is thread-safe, which makes it useful for multi-threaded programs.
- When performance is critical and thread safety is not required, use StringBuilder instead.
- Default capacity is 16, but it increases dynamically when needed.

NUMBER CLASS IN JAVA

In Java, the `java.lang.Number` class is an abstract class that serves as the superclass for all the numeric wrapper classes such as Integer, Double, Float, Long, Short, and Byte. It provides a common interface for handling numeric values in a consistent way.

- Abstract Class:
 - Cannot be instantiated directly but provides methods that are inherited by subclasses.
- Wrapper Classes:
 - The following classes extend Number:
 - Integer (for int values)
 - Double (for double values)
 - Float (for float values)
 - Long (for long values)
 - Short (for short values)
 - Byte (for byte values)
- Immutable Objects:
 - Like all wrapper classes in Java, the Number subclasses are immutable.

Common Methods of Number Class

The Number class defines several abstract methods that must be implemented by its subclasses:

Method	Description	Example
intValue()	Converts the number to an int.	num.intValue() → 10
longValue()	Converts the number to a long.	num.longValue() → 10L
floatValue()	Converts the number to a float.	num.floatValue() → 10.0f
doubleValue()	Converts the number to a double.	num.doubleValue() → 10.0
byteValue()	Converts the number to a byte.	num.byteValue() → 10
shortValue()	Converts the number to a short.	num.shortValue() → 10

Example Usage of the Number Class

```

public class NumberExample {
    public static void main(String[] args) {
        Number num1 = new Integer(100); // Integer extends Number
        Number num2 = new Double(55.5); // Double extends Number

        // Convert to different types using Number methods
        System.out.println("num1 as int: " + num1.intValue());
        System.out.println("num2 as double: " + num2.doubleValue());
        System.out.println("num1 as float: " + num1.floatValue());
        System.out.println("num2 as long: " + num2.longValue());
    }
}

```

Difference Between Number and Primitive Data Types

Feature	Primitive Types (int, double, etc.)	Number Wrapper Classes
Memory Usage	Low	Higher (object overhead)
Performance	Fast	Slightly slower
Mutability	Mutable	Immutable
Null Values	Not allowed	Allowed (null values)

The `java.util` package in Java is part of the standard library and provides a set of utility classes that support various functionalities such as data structures, date and time management, random number generation, and more. It is one of the most commonly used packages in Java development.

- Collection Framework
 - Provides interfaces and classes to store and manipulate groups of objects, such as:
 - Interfaces: List, Set, Queue, Map
 - Classes:
 - ArrayList, LinkedList (implementing List)
 - HashSet, TreeSet (implementing Set)
 - HashMap, TreeMap (implementing Map)
 - PriorityQueue (implementing Queue)
- Date and Time Handling
 - Date: Represents date and time.
 - Calendar: Provides date/time manipulation functionality.
 - TimeZone: Handles time zone-related operations.
 - GregorianCalendar: Concrete implementation of Calendar.
- Random Number Generation
 - Random: Generates random numbers of various types (int, double, etc.).
 - SecureRandom: Provides cryptographically strong random number generation.
- Utility Classes
 - Collections: Provides static methods for manipulating collections (sorting, searching, etc.).
 - Arrays: Contains utility methods for working with arrays (sorting, filling, comparing).
 - Objects: Utility methods for working with object references (null checks, comparisons).
 - StringTokenizer: Splits a string into tokens based on delimiters.
- Concurrent Utilities
 - Timer and TimerTask: Allow scheduling of tasks at fixed rates or delays.
 - BitSet: Represents a set of bits for efficient storage and manipulation.
- Scanner for Input Handling
 - Scanner: Used to parse primitive types and strings from input sources like console, files, etc.

JAVA.UTIL.RANDOM CLASS

The `java.util.Random` class in Java is used to generate pseudo-random numbers of different data types such as int, double, long, float, and boolean. It is part of the `java.util` package and provides a convenient way to create random values for use in applications such as simulations, games, and cryptography.

Key Features of Random Class:

- Pseudo-random number generation: Generates a sequence of numbers that appear random but are determined by an initial seed value.
- Supports multiple data types: Provides methods to generate random values of type int, double, float, long, boolean, etc.

- Optional seed value: If initialized with a seed, the sequence of random numbers will be repeatable.

Constructors of Random Class:

- Random()
 - Creates a new random number generator with a random seed based on the system time.
- Random(long seed)
 - Creates a new random number generator using the specified seed, ensuring reproducible sequences.

Common Methods of Random Class:

Method	Description
nextInt()	Returns a random integer (positive and negative).
nextInt(int bound)	Returns a random integer between 0 (inclusive) and bound (exclusive).
nextDouble()	Returns a random double between 0.0 and 1.0.
nextFloat()	Returns a random float between 0.0 and 1.0.
nextLong()	Returns a random long value.
nextBoolean()	Returns a random boolean value (true or false).
nextBytes(byte[] bytes)	Fills a byte array with random bytes.
setSeed(long seed)	Resets the random number generator with the given seed.

Example Usage:

```
import java.util.Random;

public class RandomExample {
    public static void main(String[] args) {
        // Create a Random object without a seed (uses system time)
        Random random = new Random();

        // Generate random integers
        System.out.println("Random Integer: " + random.nextInt());
        System.out.println("Random Integer (0 to 100): " + random.nextInt(101));

        // Generate random floating-point numbers
        System.out.println("Random Double: " + random.nextDouble());
        System.out.println("Random Float: " + random.nextFloat());

        // Generate random long value
        System.out.println("Random Long: " + random.nextLong());

        // Generate random boolean values
        System.out.println("Random Boolean: " + random.nextBoolean());

        // Generate random bytes
        byte[] bytes = new byte[5];
        random.nextBytes(bytes);
        System.out.print("Random Bytes: ");
        for (byte b : bytes) {
            System.out.print(b + " ");
        }
    }
}
```

```
}  
}
```

Use Cases of Random Class:

- Generating test data for applications.
- Simulating dice rolls, random selections, and lottery systems.
- Randomizing order of elements in an array.
- Generating random delays or time intervals.
- Creating non-secure random values for games or general-purpose applications.

JAVA.UTIL DATE CLASS

The `java.util.Date` class in Java represents a specific moment in time, with millisecond precision. It is part of the `java.util` package and is widely used for working with date and time values.

Key Features of Date Class:

- Represents date and time in milliseconds from January 1, 1970, 00:00:00 GMT.
- Provides methods to retrieve and modify date components.
- Often used in legacy code but is largely replaced by the `java.time` package (Java 8+).

Constructors of Date Class:

`Date()`

Creates a `Date` object representing the current system date and time.

```
Date currentDate = new Date();  
System.out.println(currentDate);
```

`Date(long milliseconds)`

```
Date specificDate = new Date(1000); // Example timestamp  
System.out.println(specificDate);
```

Common Methods of Date Class

Method	Description
<code>getTime()</code>	Returns the number of milliseconds since the epoch.
<code>setTime(long time)</code>	Sets the date based on milliseconds since the epoch.
<code>toString()</code>	Converts the date to a human-readable string.
<code>before(Date date)</code>	Checks if the date is before the given date.
<code>after(Date date)</code>	Checks if the date is after the given date.
<code>equals(Object obj)</code>	Compares if two <code>Date</code> objects are equal.

Example Usage:

```
import java.util.Date;

public class DateExample {
    public static void main(String[] args) {
        // Get current date and time
        Date currentDate = new Date();
        System.out.println("Current Date: " + currentDate);

        // Get time in milliseconds
        long millis = currentDate.getTime();
        System.out.println("Milliseconds since Jan 1, 1970: " + millis);

        // Set a new date using milliseconds
        Date newDate = new Date(1700000000000L);
        System.out.println("Specific Date: " + newDate);

        // Comparing dates
        Date date1 = new Date(1699999999999L);
        if (newDate.after(date1)) {
            System.out.println("newDate is after date1");
        } else {
            System.out.println("newDate is before date1");
        }
    }
}
```

JAVA.UTIL GREGORIAN CALENDAR CLASS

The `java.util.GregorianCalendar` class in Java is a concrete subclass of the `Calendar` class and provides support for the Gregorian calendar system, which is the most widely used civil calendar today. It includes support for date operations such as addition, comparison, and extraction of date fields.

Key Features of `GregorianCalendar`:

- Supports both BC (Before Christ) and AD (Anno Domini) eras.
- Provides methods to get and set individual date fields (year, month, day, etc.).
- Handles leap years automatically.
- Can be initialized with the current date and time or a custom date.
- Provides methods to add or roll date fields (e.g., add days, months, years).

Constructors of `GregorianCalendar`:

`GregorianCalendar()`

Creates a calendar initialized with the current date and time.

```
GregorianCalendar calendar = new GregorianCalendar();
```

`GregorianCalendar(int year, int month, int day)`

Creates a calendar with the specified year, month (0-based), and day.

```
GregorianCalendar calendar = new GregorianCalendar(2024, Calendar.OCTOBER, 20);
```

GregorianCalendar(int year, int month, int day, int hour, int minute)

Creates a calendar with date and time.

```
GregorianCalendar calendar = new GregorianCalendar(2024, Calendar.OCTOBER, 20, 10, 30);
```

Common Methods of GregorianCalendar:

Method	Description
get(int field)	Returns the value of the given field (e.g., year, month).
set(int field, int value)	Sets a specific field value (e.g., month to October).
add(int field, int amount)	Adds/subtracts amount from a date field.
roll(int field, int amount)	Rolls the field value without affecting larger fields.
getTime()	Returns the Date object representing the calendar date.
setTime(Date date)	Sets the calendar to a specific Date.
isLeapYear(int year)	Checks if the given year is a leap year.

Example

```
import java.util.GregorianCalendar;
import java.util.Calendar;

public class GregorianCalendarExample {
    public static void main(String[] args) {
        // Create a GregorianCalendar with the current date and time
        GregorianCalendar calendar = new GregorianCalendar();

        System.out.println("Current Date: " + calendar.getTime());

        // Get individual fields
        int year = calendar.get(Calendar.YEAR);
        int month = calendar.get(Calendar.MONTH); // Note: Month is 0-based
        int day = calendar.get(Calendar.DAY_OF_MONTH);

        System.out.println("Year: " + year);
        System.out.println("Month: " + (month + 1)); // Add 1 to month for correct display
        System.out.println("Day: " + day);
    }
}
```

Setting and modifying a date:

```
import java.util.GregorianCalendar;
import java.util.Calendar;

public class SetDateExample {
    public static void main(String[] args) {
        // Set a custom date (20th October 2024)
        GregorianCalendar calendar = new GregorianCalendar(2024, Calendar.OCTOBER, 20);

        System.out.println("Original Date: " + calendar.getTime());

        // Modify the date by adding 10 days
        calendar.add(Calendar.DAY_OF_MONTH, 10);
        System.out.println("After Adding 10 Days: " + calendar.getTime());

        // Roll the month (does not affect the year)
        calendar.roll(Calendar.MONTH, 2);
        System.out.println("After Rolling Month by 2: " + calendar.getTime());
    }
}
```

Checking Leap Year:

```
import java.util.GregorianCalendar;

public class LeapYearExample {
    public static void main(String[] args) {
        GregorianCalendar calendar = new GregorianCalendar();

        int year = 2024;
        if (calendar.isLeapYear(year)) {
            System.out.println(year + " is a leap year.");
        } else {
            System.out.println(year + " is not a leap year.");
        }
    }
}
```

Conclusion:

Use `GregorianCalendar` for backward compatibility with older Java versions or legacy systems.

Prefer `java.time` classes (like `LocalDate`, `LocalDateTime`) for new projects due to their simplicity and thread safety.

```
import java.time.LocalDate;

public class ModernDateExample {
    public static void main(String[] args) {
        LocalDate date = LocalDate.of(2024, 10, 20);
        System.out.println("Modern Date API: " + date);
    }
}
```

In Java, the Vector class is part of the java.util package and represents a growable array of objects. It implements the List interface and provides methods to dynamically grow or shrink the size of the array as elements are added or removed. The Vector class is similar to ArrayList, but it is synchronized, meaning it is thread-safe.

Here are some important aspects of the Vector class:

Key Features:

- **Dynamic Resizing:** Vector automatically increases its size when elements are added beyond its initial capacity. The default capacity is 10, and when this capacity is exceeded, it grows by doubling its size.
- **Thread-Safety:** Vector is synchronized, which makes it thread-safe but can also make it less efficient in single-threaded scenarios compared to ArrayList.
- **Index-Based Access:** It allows accessing elements by their index, just like an array.
- **Implements List Interface:** It supports operations like adding, removing, and accessing elements, similar to other List implementations.

Important Methods:

- **add(E e):** Adds an element to the vector.
- **addElement(E obj):** Adds an element to the vector (legacy method, used with older Java versions).
- **get(int index):** Returns the element at the specified index.
- **remove(int index):** Removes the element at the specified index.
- **size():** Returns the number of elements in the vector.
- **capacity():** Returns the current capacity of the vector.
- **trimToSize():** Resizes the vector to the number of elements it contains, freeing any unused space.
- **clear():** Removes all elements from the vector.

Example

```
import java.util.*;

public class VectorExample {
    public static void main(String[] args) {
        // Creating a vector with initial capacity 5
        Vector<String> vector = new Vector<>(5);
        // Adding elements to the vector
        vector.add("Java");
        vector.add("Python");
        vector.add("C++");
        // Accessing elements
        System.out.println("Element at index 1: " + vector.get(1)); // Output: Python
        // Removing an element
        vector.remove(2); // Removes "C++"

        // Printing the size and capacity of the vector
        System.out.println("Size of vector: " + vector.size()); // Output: 2
        System.out.println("Capacity of vector: " + vector.capacity()); // Output: 5

        // Trimming the capacity
        vector.trimToSize();
        System.out.println("Capacity after trim: " + vector.capacity()); // Output: 2
    }
}
```

JAVA UTIL HASHTABLE CLASS

The Hashtable class in Java, part of the java.util package, is a legacy data structure used to store key-value pairs. It is synchronized, making it thread-safe, but less efficient than newer classes like HashMap in single-threaded environments.

The Hashtable class in Java, part of the java.util package, is a legacy data structure used to store key-value pairs. It is synchronized, making it thread-safe, but less efficient than newer classes like HashMap in single-threaded environments.

Key Characteristics of Hashtable:

- Thread-Safe: All its methods are synchronized, meaning multiple threads can safely access it.
- No null Keys or Values: Unlike HashMap, it does not allow null keys or null values.
- Implementation: Implements the Map interface and is based on a hash table data structure.
- Legacy Class: It has been part of Java since JDK 1.0. For modern development, HashMap or ConcurrentHashMap is preferred.

Important Methods:

- put(K key, V value): Inserts a key-value pair into the table.
- get(Object key): Retrieves the value associated with the key.
- remove(Object key): Removes the key-value pair for the specified key.
- containsKey(Object key): Checks if the table contains a specific key.
- containsValue(Object value): Checks if the table contains a specific value.
- isEmpty(): Returns true if the table has no elements.
- size(): Returns the number of key-value pairs in the table.
- keys(): Returns an enumeration of the keys in the table.

Example: Using Hashtable

```
import java.util.Hashtable;

public class HashtableExample {
    public static void main(String[] args) {
        // Creating a Hashtable
        Hashtable<Integer, String> table = new Hashtable<>();

        // Adding key-value pairs
        table.put(1, "Java");
        table.put(2, "Python");
        table.put(3, "C++");

        // Accessing values
        System.out.println("Value for key 2: " + table.get(2)); // Output: Python

        // Removing a key-value pair
        table.remove(3);

        // Checking if a key or value exists
        System.out.println("Contains key 1? " + table.containsKey(1)); // Output: true
        System.out.println("Contains value 'C++'? " + table.containsValue("C++")); // Output:
false
```

```

// Printing all key-value pairs
System.out.println("All entries: " + table);

// Iterating through keys
System.out.println("Keys in the Hashtable:");
for (Integer key : table.keySet()) {
    System.out.println("Key: " + key + ", Value: " + table.get(key));
}
}
}

```

StringTokenizer

The StringTokenizer class in Java (part of the java.util package) is a legacy utility for breaking strings into tokens. It is used to split a string into smaller parts (tokens) based on specified delimiters.

While still available, it's generally recommended to use modern alternatives like the split() method of the String class or Scanner for more flexibility.

Constructor Variants:

- StringTokenizer(String str): Uses the default delimiter (space, \t, \n, \r, \f).
- StringTokenizer(String str, String delimiter): Splits the string using the specified delimiter.
- StringTokenizer(String str, String delimiter, boolean returnDelims): If returnDelims is true, the delimiters are treated as tokens as well.

Key Methods:

- boolean hasMoreTokens(): Checks if there are more tokens available.
- String nextToken(): Returns the next token from the string.
- String nextToken(String delimiter): Returns the next token using a new delimiter.
- int countTokens(): Returns the number of tokens available.

Example: Using StringTokenizer

```

import java.util.StringTokenizer;

public class StringTokenizerExample {
    public static void main(String[] args) {
        // Input string
        String input = "Java,Python,C++,JavaScript";

        // StringTokenizer with comma as delimiter
        StringTokenizer tokenizer = new StringTokenizer(input, ",");

        // Iterating through tokens
        System.out.println("Tokens:");
        while (tokenizer.hasMoreTokens()) {
            System.out.println(tokenizer.nextToken());
        }

        // Using countTokens()
        StringTokenizer tokenizer2 = new StringTokenizer("One;Two;Three;Four", ";");
        System.out.println("\nNumber of tokens: " + tokenizer2.countTokens());
    }
}

```

SIMPLETIMEZONE

The `SimpleTimeZone` class in Java (part of the `java.util` package) is used to represent a time zone with support for Daylight Saving Time (DST) adjustments. It's a subclass of `TimeZone` and is considered less flexible than the newer `java.time` API introduced in Java 8. However, it remains useful in certain legacy applications.

Key Features:

- Customizable Time Zones:
- You can create a `SimpleTimeZone` by specifying the time zone offset from UTC and optionally configuring DST rules.
- Daylight Saving Time (DST):
- It allows you to define rules for when DST starts and ends.
- Legacy Class:
- It's a part of the older `java.util` package.
- For modern applications, the `java.time.ZoneId` or `java.time.ZonedDateTime` is preferred.

Constructors:

The most common constructor for `SimpleTimeZone` is:

`SimpleTimeZone(int rawOffset, String ID)`

Example: Creating a `SimpleTimeZone`

```
import java.util.SimpleTimeZone;

public class SimpleTimeZoneExample {
    public static void main(String[] args) {
        // Create a time zone for GMT+5:30 (IST)
        int rawOffset = 5 * 60 * 60 * 1000 + 30 * 60 * 1000; // 5 hours 30 minutes in milliseconds
        SimpleTimeZone timeZone = new SimpleTimeZone(rawOffset, "IST");

        // Print details
        System.out.println("TimeZone ID: " + timeZone.getID());
        System.out.println("Raw Offset (in hours): " + (timeZone.getRawOffset() / (60 * 60 * 1000)) + " hours");
    }
}
```

What is Daylight Saving Time (DST)?

Daylight Saving Time (DST) is the practice of setting clocks forward by one hour during the warmer months of the year (typically spring and summer) to extend evening daylight and conserve energy. At the end of the season (typically fall or winter), clocks are set back by one hour to return to standard time.

STACK

The `Stack` class in Java (part of the `java.util` package) is a last-in, first-out (LIFO) data structure that models a collection of elements where the last element added is the first one to be removed. The

class is a direct subclass of Vector and is part of the legacy collection framework, which means it has been largely replaced by more modern classes like Deque (e.g., ArrayDeque) in newer Java versions. However, Stack is still useful in certain scenarios.

Key Features of Stack:

- LIFO Order: The most recently added element is removed first.
- Thread-Safety: Similar to Vector, it is synchronized, making it thread-safe but less efficient than other alternatives in single-threaded applications.
- Inheritance: It extends Vector, which means it has all the functionality of a Vector along with additional methods specific to stack operations.

Commonly Used Methods:

- push(E item): Pushes an element onto the top of the stack.
- pop(): Removes and returns the element at the top of the stack.
- peek(): Returns the element at the top of the stack without removing it.
- empty(): Checks if the stack is empty. Returns true if empty, false otherwise.
- search(Object o): Returns the 1-based position of the element from the top of the stack. Returns -1 if the element is not found.

Example: Using Stack Class

```
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        // Creating a Stack of Strings
        Stack<String> stack = new Stack<>();

        // Pushing elements onto the stack
        stack.push("Java");
        stack.push("Python");
        stack.push("C++");
        stack.push("JavaScript");

        // Peeking at the top element (without removing it)
        System.out.println("Top element: " + stack.peek()); // Output: JavaScript

        // Popping elements from the stack (removes the top element)
        System.out.println("Popped element: " + stack.pop()); // Output: JavaScript
        System.out.println("Popped element: " + stack.pop()); // Output: C++

        // Checking if the stack is empty
        System.out.println("Is stack empty? " + stack.empty()); // Output: false

        // Searching for an element
        int position = stack.search("Python");
        System.out.println("Position of 'Python': " + position); // Output: 1 (from top of
stack)

        // Remaining elements in the stack
        System.out.println("Remaining elements in stack: " + stack);
    }
}
```

SCANNER

The Scanner class in Java, part of the `java.util` package, is used to read input from various sources like keyboard input, files, or strings. It makes it easy to parse primitive types (e.g., `int`, `float`, `double`, `String`) from input streams.

Key Features of the Scanner Class:

- **Input Sources:** It can read input from different sources such as:
 - **Keyboard:** `System.in` (standard input).
 - **Files:** You can pass a `File` object to read from a file.
 - **Strings:** It can read from a string, using `StringReader`.
- **Flexible Parsing:** It can parse tokens (tokens are whitespace-separated strings) and convert them into various primitive types.
- **Regex Support:** Allows you to specify delimiters using regular expressions.

Commonly Used Methods of the Scanner Class:

- `next()`: Returns the next token as a string.
- `nextInt()`: Returns the next token as an integer.
- `nextDouble()`: Returns the next token as a double.
- `nextLine()`: Returns the next line of input as a string.
- `hasNext()`: Returns true if there is another token to read.
- `hasNextInt()`, `hasNextDouble()`, etc.: Checks if the next token can be interpreted as the specified type.

Example: Using Scanner for Keyboard Input

```
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        // Create a Scanner object to read input from the console (keyboard)
        Scanner scanner = new Scanner(System.in);

        // Reading a String
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name + "!");

        // Reading an integer
        System.out.print("Enter your age: ");
        int age = scanner.nextInt();
        System.out.println("Your age is: " + age);

        // Reading a double (e.g., for height or weight)
        System.out.print("Enter your height in meters: ");
        double height = scanner.nextDouble();
        System.out.println("Your height is: " + height + " meters");

        // Closing the scanner to free resources
        scanner.close();
    }
}
```

Explanation of Methods:

- `nextLine()`: This reads a full line of input until the user hits "Enter".
- `nextInt()`: Reads the next integer token from the input.
- `nextDouble()`: Reads the next double token from the input.

Example: Using Scanner to Read from a File

You can also use Scanner to read from a file:

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class FileScannerExample {
    public static void main(String[] args) {
        try {
            // Create a Scanner object to read from a file
            File file = new File("example.txt");
            Scanner scanner = new Scanner(file);

            // Read each line from the file
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                System.out.println(line);
            }

            // Closing the scanner
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found.");
        }
    }
}
```

Example: Using Custom Delimiters

You can use Scanner with custom delimiters. By default, it uses whitespace (\s) as the delimiter. You can change this to a custom delimiter like a comma, period, etc.

```
import java.util.Scanner;

public class CustomDelimiterExample {
    public static void main(String[] args) {
        String input = "apple,banana,orange,grape";

        // Create a Scanner with a custom delimiter (comma)
        Scanner scanner = new Scanner(input);
        scanner.useDelimiter(",");

        // Read and print tokens separated by commas
        while (scanner.hasNext()) {
            System.out.println(scanner.next());
        }

        scanner.close();
    }
}
```

CREATING AND USING USER DEFINED PACKAGE AND SUB-PACKAGE, IN JAVA

In Java, a package is a namespace that organizes classes, interfaces, and sub-packages. It is used to group related classes and avoid name conflicts. Packages in Java are similar to folders in a file system—they help in categorizing classes logically and manage them efficiently.

Why Use Packages?

- **Organize Code Logically:** Helps group related classes and interfaces for better readability and management.
- **Avoid Name Conflicts:** Classes with the same name in different packages can coexist without clashes.
- **Code Reusability:** Classes in a package can be reused in other programs.
- **Access Control:** Packages allow defining class access levels (e.g., public, protected, default).
- **Encapsulation:** Helps in grouping related classes for a module and hiding implementation details.

In Java, user-defined packages and sub-packages help organize classes and interfaces logically and modularly. Here's how you can create and use them:

Types of Packages in Java

Java has two types of packages:

Built-in Packages: These are provided by Java and include commonly used libraries.

Examples: java.util, java.io, java.net, java.sql, java.lang, etc.

User-Defined Packages: These are created by the programmer to organize custom classes and interfaces.

package com.myapp.util; // Custom package declaration

How to Create and Use Packages?

1. Create a Package

Use the package keyword at the top of your Java file:

```
package mypackage;

public class MyClass {
    public void greet() {
        System.out.println("Hello from MyClass!");
    }
}
```

Compile a Package

javac -d . MyClass.java

Use a Package

Import and use the class in another program:

```
import mypackage.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.greet();
    }
}
```

```
}  
}
```

Run the Program

After compiling, run the main program using:

java Main

EXCEPTION HANDLING:

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing program. A mistake might lead to an error causing the program to produce unexpected results. Errors are the wrongs that can make a program go wrong

An error may produce an incorrect output or may terminate the execution of the program unexpectedly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

TYPE OF ERRORS

Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

COMPILE-TIME ERRORS

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the .class file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

```
/*This program contains an error*/  
class error1  
{  
    public static void main(String args[])  
    {  
        System.out.println("Hello friends") //missing ;  
    }  
}
```

RUN-TIME ERRORS

Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack Overflow. Most common run-time errors are:

- Dividing an integer by zero.
- Accessing an element that is out of the bounds of an array.
- Trying to store a value into an array of an incompatible class or type.
- Trying to cast an instance of a class to one of its subclasses.
- Passing a parameter that is not in a valid range of value for a method.
- Attempting to use a negative size for an array.
- Converting invalid string to a number

- Accessing a character that is out of bounds of a string and many more.

EXCEPTION

The Exception Handling in Java is one of the powerful mechanisms to handle the runtime errors so that the normal flow of the application can be maintained.

What is Exception in Java?

In Java, an exception is an event that occurs during the execution of a program that interrupt the normal flow of instructions. These exceptions can occur for various reasons, such as invalid user input, file not found, or division by zero. When an exception occurs, it is typically represented by an object of a subclass of the `java.lang.Exception` class.

An exception in Java is an unexpected event or error that occurs during the execution of a program, disrupting the normal flow of instructions. It is an object that represents runtime errors such as divide by zero, accessing an array out of bounds, or attempting to use a null reference.

Java provides a robust exception-handling mechanism to detect and handle such errors gracefully, ensuring the program does not crash unexpectedly.

EXCEPTION HANDLING MECHANISM

Java provides the `try`, `catch`, `finally`, `throw`, and `throws` keywords for handling exceptions.

Example:

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int a = 10 / 0; // ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        } finally {
            System.out.println("Execution completed.");
        }
    }
}
```

Java Exception Keywords

Keyword	Description
Try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
Catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
Finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
Throw	The "throw" keyword is used to throw an exception.

Throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.
--------	---

The try-catch Block

One of the primary mechanisms for handling exceptions in Java is the try-catch block. The try block contains the code that may throw an exception, and the catch block is used to handle the exception if it occurs. Here's a basic example:

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Exception handling code
}
```

HANDLING MULTIPLE EXCEPTIONS

You can handle multiple types of exceptions by providing multiple catch blocks, each catching a different type of exception. This allows you to tailor your exception handling logic based on the specific type of exception thrown. Here's an example:

```
try {
    // Code that may throw an exception
} catch (IOException e) {
    // Handle IOException
} catch (NumberFormatException e) {
    // Handle NumberFormatException
} catch (Exception e) {
    // Handle any other exceptions
}
```

THE FINALLY BLOCK

In addition to try and catch, Java also provides a finally block, which allows you to execute cleanup code, such as closing resources, regardless of whether an exception occurs or not. The finally block is typically used to release resources that were acquired in the try block. Here's an example:

```
try {
    // Code that may throw an exception
} catch (Exception e) {
    // Exception handling code
} finally {
    // Cleanup code
}
```

JAVA EXCEPTION HANDLING KEYWORDS

1. try

The try block is used to write the code that may cause an exception.

If an exception occurs, it is caught by the catch block.

If no exception occurs, the catch block is skipped.

```
try {  
    int a = 10 / 0; // This will cause an exception  
}
```

2. catch

The catch block is used to handle the exception.

It runs only if an exception occurs in the try block.

```
try {  
    int a = 10 / 0; // Error occurs here  
} catch (ArithmeticException e) {  
    System.out.println("Cannot divide by zero!"); // Handles the error  
}
```

3. finally

The finally block runs always, whether an exception occurs or not.

It is usually used to release resources (like closing files or database connections).

```
try {  
    int a = 10 / 0;  
} catch (ArithmeticException e) {  
    System.out.println("Error: " + e.getMessage());  
} finally {  
    System.out.println("This will always run!"); // Executes no matter what  
}
```

4. throw

The throw keyword is used to manually throw an exception.

throw new ArithmeticException("Custom error message");

```
public class Example {  
    static void checkAge(int age) {  
        if (age < 18) {  
            throw new ArithmeticException("Not eligible to vote");  
        }  
    }  
}
```

```

    }
}
public static void main(String[] args) {
    checkAge(16); // This will throw an exception
}
}

```

5. throws

The throws keyword is used in method signatures to indicate that a method might throw an exception.

It does not handle the exception; it just informs that an exception might occur.

```

public void readFile() throws IOException {
    // Code that may cause IOException
}

```

```

import java.io.IOException;
public class Example {
    static void testMethod() throws IOException {
        throw new IOException("IO error occurred");
    }

    public static void main(String[] args) {
        try {
            testMethod();
        } catch (IOException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}

```

Practical lab exercises on Exception Handling

- Write a Java program to take two numbers as input from the user and perform division. Handle the `ArithmeticException` if the user enters zero as the denominator.
- Write a Java program to take an array of 5 elements. Ask the user for an index and print the element at that index. Handle `ArrayIndexOutOfBoundsException` if the index is out of range.
- Write a Java program to accept two numbers from the user and divide them. Handle both `ArithmeticException` and `InputMismatchException` (if the user enters a non-numeric value).
- Write a Java program where a user inputs their age. If the age is less than 18, throw an exception manually (throw new `Exception`). Handle the exception in a try-catch block.
- Create a user-defined exception class called `InvalidAgeException`. In the `main()` method, ask for the user's age. If age is below 18, throw `InvalidAgeException`, otherwise print "You are eligible to vote."

- Create a BankAccount class with deposit and withdrawal methods. Define a custom exception InsufficientBalanceException. If a user tries to withdraw more than the available balance, throw this exception.

THREADS

Before introducing the thread concept, we were unable to run more than one task in parallel. It was a drawback, and to remove that drawback, Thread Concept was introduced.

Definition & Concept : In Java, a thread is the smallest unit of execution within a process. Threads allow a program to operate more efficiently by performing multiple tasks simultaneously. Each thread runs in a separate path of execution, enabling concurrent execution of code segments. This is particularly useful for tasks that can be parallelized, such as handling multiple user requests, performing background tasks, or improving the responsiveness of graphical user interfaces (GUIs).

A Thread is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

Creating a Thread:

There are two ways to create a thread in JAVA, by extending thread class and by implementing runnable interface.

Thread Methods:

start():

Starts the thread and calls the run() method.

The thread transitions from the "New" state to the "Runnable" state.

```
Thread thread = new Thread(() -> System.out.println("Thread is running."));
```

```
thread.start();
```

run():

Contains the code that the thread executes when it starts.

Override this method when extending the Thread class or implementing the Runnable interface.

```
class MyThread extends Thread {
```

```
    @Override
```

```

public void run() {
    System.out.println("Thread is running.");
}
}

```

sleep(long millis):

Pauses the execution of the current thread for the specified number of milliseconds.

Throws InterruptedException if the thread is interrupted while sleeping.

```

try {
    Thread.sleep(1000); // Sleep for 1 second
} catch (InterruptedException e) {
    System.out.println("Thread interrupted.");
}

```

join():

Waits for the thread to terminate.¹

Useful for ensuring that one thread completes before another proceeds.

```
Thread thread = new Thread(() -> System.out.println("Thread is running."));
```

```
thread.start();
```

```
thread.join(); // Wait for the thread to finish
```

```
System.out.println("Main thread continues after join.");
```

yield():

Pauses the current thread and allows other threads to execute.

This is a hint to the thread scheduler but does not guarantee that other threads will run.

```
Thread.yield(); // Pause current thread
```

isAlive():

Checks if the thread is still running.

Returns true if the thread is in the "Runnable" or "Running" state, otherwise false.

```
Thread thread = new Thread(() -> System.out.println("Thread is running."));
```

```
thread.start();
```

```
System.out.println("Is thread alive? " + thread.isAlive());
```

setName(String name) and getName():

setName(String name): Sets the name of the thread.

getName(): Returns the name of the thread.

```
Thread thread = new Thread(() -> System.out.println("Thread is running."));
```

```
thread.setName("MyThread");
```

```
System.out.println("Thread name: " + thread.getName());
```

Creating a Thread by Extending Thread class:

```
// Step 1: Extend the Thread class
class MyThread extends Thread {
    // Step 2: Override the run() method
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - Count: " + i);
            try {
                Thread.sleep(500); // Simulate some work with a sleep
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted.");
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Step 3: Create an instance of the custom thread class
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();

        // Step 4: Start the threads using the start() method
        thread1.start(); // Starts thread1
        thread2.start(); // Starts thread2

        // The main thread continues executing
        System.out.println("Main thread finished.");
    }
}
```

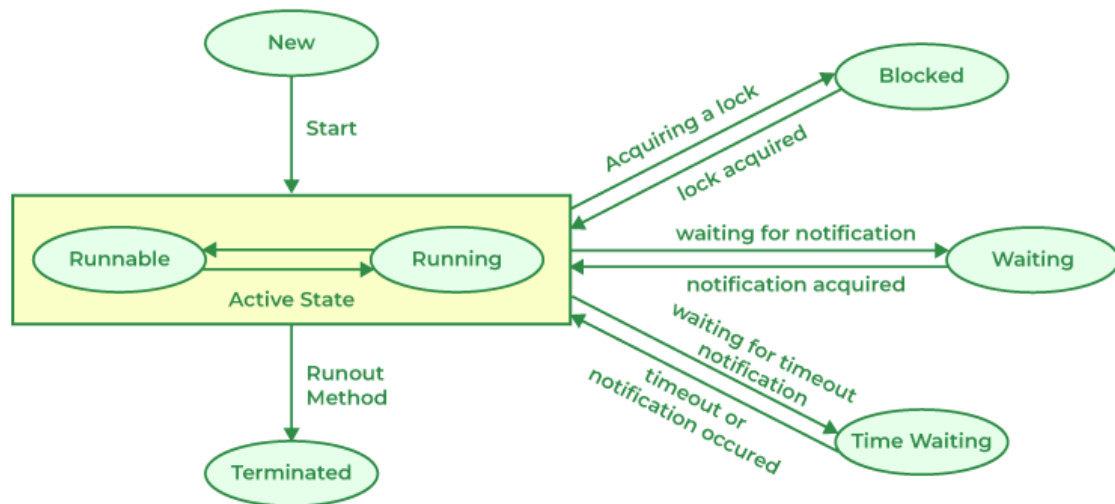
```
class mthread1 extends Thread
```

```

{
    public void run()
    {
        while(true)
        {
            System.out.println("I am Eating apple");
        }
    }
}
class mthread2 extends Thread
{
    public void run()
    {
        while(true)
        {
            System.out.println("I am Eating Bycycle");
        }
    }
}
class ethread
{
    public static void main(String[] args)
    {
        mthread1 t1 = new mthread1();
        mthread2 t2 = new mthread2();

        t1.start();
        t2.start();
    }
}

```



A thread in Java can exist in any one of the following states at any given time. A thread lies only in one of the shown states at any instant:

- New State
- Runnable State
- Blocked State
- Waiting State
- Timed Waiting State
- Terminated State

There are multiple states of the thread in a lifecycle as mentioned below:

New Thread: When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.

Thread t = new Thread();

Runnable State: A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread get a small amount of time to run. After running for a while, a thread pauses and gives up the CPU so that other threads can run.

t.start(); // Transition to Runnable state

Blocked: The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.

t.wait() / t.sleep()

Waiting state: The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.

Timed Waiting: A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

Terminated State: A thread terminates because of either of the following reasons:

Because it exits normally. This happens when the code of the thread has been entirely executed by the program.

Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

Summary of Thread Life Cycle:

New → Runnable → (Possibly through Blocked, Waiting, or Timed Waiting) → Terminated

- Write a Java program to create a basic Java thread that prints "Hello, World!" when executed
- Write a Java program that creates two threads to find and print even and odd numbers from 1 to 20.

JOIN() METHOD:

In Java, the main thread is the primary thread that starts when a Java program runs. It's created by the Java Virtual Machine (JVM) and is responsible for running the code inside the main() method.

The main thread is the entry point for every Java application.

It's the parent thread of any other threads you create in your program.

If the main thread finishes, the program will terminate — unless other non-daemon threads are still running.

You can see the main thread in action like this:

```
public class MainThreadExample {
    public static void main(String[] args) {
        // Getting reference to the main thread
        Thread mainThread = Thread.currentThread();
    }
}
```

```

        System.out.println("Main thread: " + mainThread.getName());
        System.out.println("Main thread priority: " + mainThread.getPriority());
        System.out.println("Main thread state: " + mainThread.getState());
    }
}

```

Here's a simple example of using the join() method in Java threads

```

public class Testjoin extends Thread
{
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("Child Thread here" + i);
                Thread.sleep(1000);
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }

    public static void main(String[] args) //throws InterruptedException
    {
        Testjoin t=new Testjoin();
        t.start();
        //t.join();
        try
        {
            for(int i=1;i<=5;i++)
            {
                System.out.println("Main Thread here" + i);
                Thread.sleep(1000);
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}

```

HERE'S A SIMPLE EXAMPLE OF THE YIELD() METHOD IN JAVA:

```

class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - Count: " + i);

            // Suggesting the scheduler to pause and let other threads execute
            Thread.yield();
        }
    }
}

public class YieldExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        t2.start();
    }
}

```

```
}
```

How `yield()` works:

`Thread.yield()` tells the thread scheduler that the current thread is willing to pause and let other threads of equal priority run.

It doesn't guarantee that the thread will yield immediately — the thread scheduler may or may not pause the current thread.

It's mainly used for testing and debugging when you want to check how threads interact.

`isAlive()`

The `isAlive()` method in Java is used to check if a thread is still active or alive. A thread is considered "alive" from the moment it's started (`start()` method called) until it finishes execution (when the `run()` method completes).

```
public final boolean isAlive()
```

It returns:

true if the thread is started and still running or in a waiting state.

false if the thread has not started yet or already finished execution.

Simple example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is running.");
        try {
            Thread.sleep(1000); // Simulate some work
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println(Thread.currentThread().getName() + " has finished.");
    }
}

public class IsAliveExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();

        System.out.println("Before starting thread: " + t1.isAlive()); // false

        t1.start();
        System.out.println("After starting thread: " + t1.isAlive()); // true

        try {
            t1.join(); // Wait for thread to finish
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }

        System.out.println("After thread finished: " + t1.isAlive()); // false
    }
}
```

Real-life Example of Java Multithreading:

Let's take a real-life example of multithreading in Java — imagine an online food delivery app like Zomato or Swiggy. Multiple tasks happen at the same time:

- Order placement
- Preparing the food
- Assigning a delivery person
- Tracking the delivery

Each of these can run as an independent thread, working in parallel without waiting for the other tasks to complete. Let's model this with Java multithreading:

```
class OrderPlacement extends Thread {
    public void run() {
        System.out.println("Order placed successfully!");
        try {
            Thread.sleep(1000); // Simulate time taken to place an order
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Order confirmed!");
    }
}

class FoodPreparation extends Thread {
    public void run() {
        System.out.println("Food preparation started...");
        try {
            Thread.sleep(2000); // Simulate time taken to prepare food
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Food is ready!");
    }
}

class AssignDelivery extends Thread {
    public void run() {
        System.out.println("Searching for a delivery partner...");
        try {
            Thread.sleep(1500); // Simulate time taken to assign a delivery partner
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Delivery partner assigned!");
    }
}

class DeliveryTracking extends Thread {
    public void run() {
        System.out.println("Tracking the delivery...");
        try {
            Thread.sleep(2500); // Simulate time taken for delivery
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Order delivered successfully!");
    }
}

public class FoodDeliveryApp {
```

```

public static void main(String[] args) {
    OrderPlacement order = new OrderPlacement();
    FoodPreparation food = new FoodPreparation();
    AssignDelivery delivery = new AssignDelivery();
    DeliveryTracking tracking = new DeliveryTracking();

    order.start();
    food.start();
    delivery.start();
    tracking.start();
}
}

```

In the above example, we come across both multiprocessing and multithreading. These are somehow indirectly used to achieve multitasking. In this way the mechanism of dividing the tasks is called multithreading in which every process or task is called by a thread where a thread is responsible for when to execute, when to stop and how long to be in a waiting state. Hence, a thread is the smallest unit of processing whereas multitasking is a process of executing multiple tasks at a time.

DAEMON THREAD

A daemon thread in Java is a special type of thread that provides background services and doesn't prevent the JVM from exiting when all non-daemon threads finish execution.

In simple words:

Daemon threads support other threads — like garbage collection or handling listeners.

They run in the background and stop automatically when all user (non-daemon) threads complete.

Examples of daemon threads:

- Garbage Collector (GC)
- Finalizer thread
- Timer threads

How to create a daemon thread:

We use the `setDaemon(true)` method before calling `start()` on a thread.

Simple example:

```

public class DThread extends Thread
{
    public void run()
    {
        //System.out.println(Thread.currentThread().isDaemon());
        System.out.println("Child Thread");
    }
    public static void main(String[] args) throws InterruptedException
    {

```

```

        System.out.println("Main Thread");
        DThread t=new DThread();
        t.setDaemon(true);
        t.start();
        Thread.sleep(1000);
    }
}

```

Non-daemon thread

A non-daemon thread (also called a user thread) in Java is a regular thread that the JVM waits for to finish before it shuts down. Unlike daemon threads, non-daemon threads keep the JVM running until they complete their execution — even if all other threads have finished.

Key characteristics of non-daemon threads:

- They perform main application tasks (like processing user requests, saving data, etc.).
- JVM waits for them to finish before exiting.
- By default, every thread you create is a non-daemon thread unless explicitly set as a daemon.

Example:

```

class UserThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " (Non-daemon) started.");
        try {
            Thread.sleep(3000); // Simulate some work
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println(Thread.currentThread().getName() + " (Non-daemon) finished.");
    }
}

public class NonDaemonThreadDemo {
    public static void main(String[] args) {
        UserThread t1 = new UserThread();
        t1.start();

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

```

Difference between Daemon and Non-Daemon Threads:

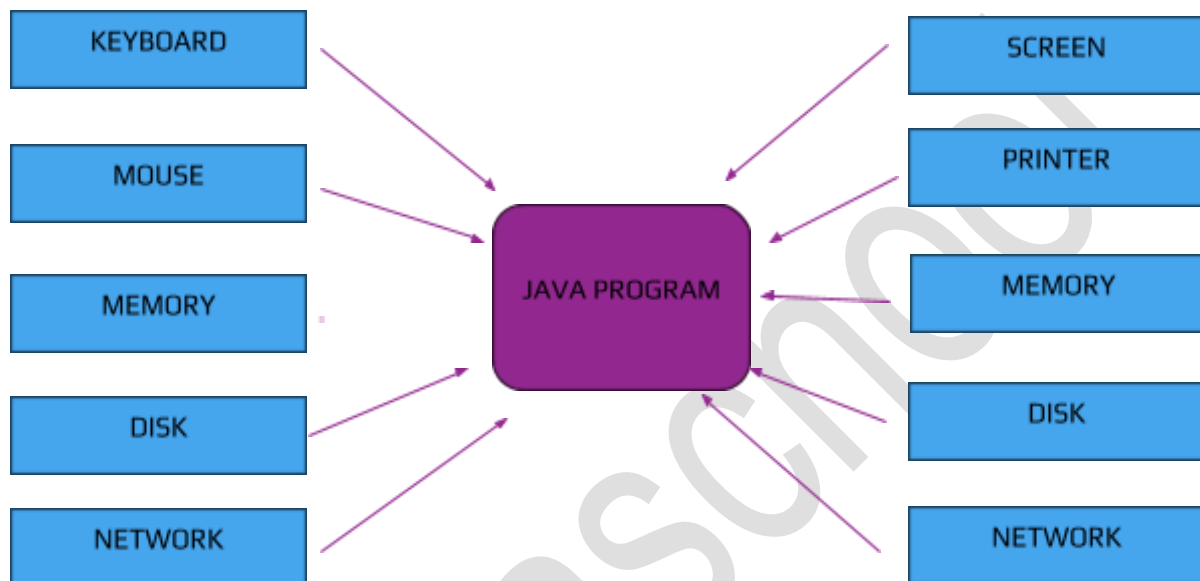
Feature	Daemon Thread	Non-Daemon Thread
Purpose	Background tasks	Main application tasks
JVM behavior	Stops when all non-daemon threads finish	JVM waits for it to finish
Default	False (must be set explicitly)	True (all threads are non-daemon by default)
Example	Garbage Collector, Listeners	Processing user requests, saving files

STREAM:

A Stream in Java is a sequence of data that flows from a source to a destination. Think of it like a pipeline where data moves step by step — either for reading (input) or writing (output). Streams don't store data; they only transfer it.

Java provides the `java.io` package, which contains classes for working with streams. These streams make it easy to perform input and output (I/O) operations, like reading from files, writing to files, or taking input from the keyboard and displaying output on the screen.

The concept of sending data from one stream to another (like one pipe feeding into another pipe) has made stream in java a powerful tool for processing.



Java streams are classified into two basic types, namely, input stream and output stream.

An input stream extracts (i.e. reads) data from the source (file) and sends in to the program.

Similarly, an output stream takes data from the program and sends(i.e. writes) it to the destination file.



Reading & Writing data in to program

Input Stream: Data flows into your program (e.g., reading a file).

Output Stream: Data flows out of your program (e.g., writing to a file).

Byte Stream

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, `FileInputStream` and `FileOutputStream`.

Example of read and write with byte by byte using FileInputStream and FileOutputStream.

```
import java.io.*;
class copyfile
{
    public static void main(String args[]) throws IOException
    {
        FileInputStream in = null;
        FileOutputStream out= null;
        try
        {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("out.txt");
            int c;
            while((c = in.read()) != -1)
            {
                out.write(c);
            }
        }
        finally
        {
            if(in != null)
            {
                in.close();
            }
            if(out != null)
            {
                out.close();
            }
        }
    }
}
```

Character Stream

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams But the Most frequently used classes are, FileReader and FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and File Writer writes two bytes at a time.

```
import java.io.*;
class copyfile1
{
    public static void main(String args[]) throws IOException
    {
        FileReader in = null;
        FileWriter out= null;
        try
        {
            in = new FileReader("input.txt");
            out = new FileWriter("out1.txt");
            int c;
            while((c = in.read()) != -1)
            {
                out.write(c);
            }
        }
    }
}
```

```

        out.write(c);
    }
    finally
    {
        if(in != null)
        {
            in.close();
        }
        if(out != null)
        {
            out.close();
        }
    }
}
}

```

```

import java.io.*;
public class reading {
    public static void main(String[] args)
    {
        // Try-catch block for exception handling
        try {

            // Create a FileReader object
            // which is a subclass of Reader
            Reader r = new FileReader("example.txt");

            // Read one character at a time from the file
            int data = r.read();
            while (data != -1) {

                // Convert the int to char and print
                System.out.print((char)data);
                data = r.read();
            }

            // Close the reader
            r.close();
        }
        catch (Exception ex) {

            // Handle any IO exceptions
            System.out.println("An error occurred: "
                               + ex.getMessage());
        }
    }
}

```

```

// Demonstrate the working of FileWriter class
import java.io.FileWriter;
import java.io.IOException;

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

            // Create a FileWriter to write to a file named
            // "example.txt"

```

```

        FileWriter w = new FileWriter("example1.txt");

        // Write a simple message to the file
        w.write("Hello, World!");

        // Close the writer
        w.close();

        System.out.println(
            "Writing Done");
    }
    catch (IOException e) {
        System.out.println("An error occurred: "
            + e.getMessage());
    }
}

```

Types of Streams:

Java mainly has two kinds of streams:

Input Streams: Used for reading data from an input source (like a file, keyboard, network, etc.).
Examples: `FileInputStream`, `BufferedInputStream`, `DataInputStream`.

Output Streams: Used for writing data to an output destination (like a file, console, network, etc.).
Examples: `FileOutputStream`, `BufferedOutputStream`, `DataOutputStream`.

Streams are also categorized based on the type of data they handle:

3. **Byte Streams:** Used for handling binary data (like images, audio, video, etc.). These read/write data byte-by-byte. Examples: `InputStream`, `OutputStream`.

4. **Character Streams:** Used for handling textual data (like .txt files). These read/write data character-by-character. Examples: `Reader`, `Writer`.

★ File Handling in Java

📘 Definition

File handling in Java is the process of creating, reading, writing, and deleting files using Java classes. It allows programs to store data permanently on secondary storage (hard disk).

📌 Why File Handling is Needed?

- To store data permanently
- To read data from files
- To write output to files
- To process large data
- Used in real applications like reports, logs, records

★ Packages Used in File Handling

Java file handling is mainly done using:

```
java.io
```

★ Important Classes Used in File Handling

Class	Purpose
File	Represents file or directory
FileWriter	Writes data to file
FileReader	Reads data from file
BufferedWriter	Writes data efficiently
BufferedReader	Reads data efficiently
FileInputStream	Reads binary data
FileOutputStream	Writes binary data

★ Basic File Operations

- ◆ 1. Create a File ◆ 2. Write to a File ◆ 3. Read from a File ◆ 4. Append data ◆ 5. Delete a File

★ Simple Example: Writing to a File

```
import java.io.FileWriter;  
  
public class WriteFile {  
    public static void main(String[] args) throws Exception {  
  
        FileWriter fw = new FileWriter("test.txt");  
        fw.write("Welcome to Java File Handling");  
        fw.close();  
  
        System.out.println("File Written Successfully");  
    }  
}
```

★ Simple Example: Reading from a File

```
import java.io.FileReader;  
  
public class ReadFile {  
    public static void main(String[] args) throws Exception {  
  
        FileReader fr = new FileReader("test.txt");  
        int ch;  
  
        while((ch = fr.read()) != -1) {  
            System.out.print((char) ch);  
        }  
        fr.close();  
    }  
}
```

★ File Class Example

```
import java.io.File;  
  
public class FileDemo {  
    public static void main(String[] args) {  
  
        File f = new File("test.txt");  
  
        System.out.println("File exists: " + f.exists());  
        System.out.println("File name: " + f.getName());  
        System.out.println("File size: " + f.length());  
    }  
}
```

★ Exception Handling in File Handling

File handling operations may cause errors, so they are handled using:

```
try { }  
catch(IOException e) { }
```

★ Advantages of File Handling

- Permanent data storage
- Easy data access
- Supports large data
- Useful for real-world applications

★ File Truncation in Java (Like C Language)

◆ What is File Truncation?

Removing all existing data from a file and making it empty, without deleting the file itself.

🔗 Same concept as C language "w" mode.

★ How File Truncation Works in Java

✅ Method 1: Using FileWriter (Most Common & Easy)

```
import java.io.FileWriter;  
  
public class TruncateFile {  
    public static void main(String[] args) throws Exception {  
  
        // This will TRUNCATE the file  
        FileWriter fw = new FileWriter("data.txt");  
        fw.close();  
  
        System.out.println("File truncated successfully");  
    }  
}
```

✓ If file exists → content deleted

✓ If file does not exist → new file created

👉 Exactly like fopen("data.txt","w") in C

✅ Method 2: Using Append Mode (To Prevent Truncation)

```
FileWriter fw = new FileWriter("data.txt", true);
```

✓ Keeps old data

✓ Writes at the end

👉 Like `fopen("data.txt", "a")` in C

★ File Open Modes: C vs Java (BEST COMPARISON)

C Mode	Meaning (C)	Java Equivalent	Effect
"r"	Read only	FileReader	Read file
"w"	Write (truncate)	FileWriter(file)	Overwrite file
"a"	Append	FileWriter(file, true)	Add data
"r+"	Read + Write	RandomAccessFile("rw")	Read & write
"w+"	Read + Write (truncate)	RandomAccessFile("rw") setLength(0)	+ Clear file
"a+"	Read + Append	RandomAccessFile("rw") + seek end	Append & read

★ Using RandomAccessFile (Advanced – Like C)

◆ Truncate File Explicitly

```
import java.io.RandomAccessFile;

public class RAFTTruncate {
    public static void main(String[] args) throws Exception {

        RandomAccessFile raf = new RandomAccessFile("data.txt", "rw");
        raf.setLength(0);    // truncate file
        raf.close();

        System.out.println("File truncated");
    }
}
```

✓ Direct truncation

✓ Similar to low-level C file handling

★ Important Java File Modes Explained

◆ FileWriter(file)

- Write mode
- Truncates file
- Like "w" in C

◆ FileWriter(file, true)

- Append mode
- Does NOT truncate
- Like "a" in C

◆ RandomAccessFile("rw")

- Read + Write
- Pointer-based access
- Similar to "r+", "w+", "a+"

File truncation in Java means clearing all data from a file. It can be achieved by opening the file in write mode using `FileWriter`, which overwrites existing content. Java file modes are similar to C file modes, such as write, append, and read-write. `RandomAccessFile` provides advanced file handling similar to C language.

★ Difference Between `FileWriter` and `BufferedWriter`

📘 `FileWriter`

Definition:

`FileWriter` is a character stream class used to write text directly to a file.

◆ Key Points

- Writes data character by character
- Directly writes to the file
- Slower for large data
- Simple to use
- Part of `java.io`

◆ Example

```
FileWriter fw = new FileWriter("data.txt");
fw.write("Hello Java");
fw.close();
```

BufferedWriter

Definition:

BufferedWriter is a character stream class that writes data using a buffer (temporary memory) to improve performance.

◆ Key Points

- Writes data in chunks (buffered)
- Uses FileWriter internally
- Faster than FileWriter
- Efficient for large data
- Requires flush() or close()

◆ Example

```
BufferedWriter bw = new BufferedWriter(new FileWriter("data.txt"));  
bw.write("Hello Java");  
bw.close();
```

★ Main Difference Table (Very Important for Exam)

FileWriter	BufferedWriter
Writes directly to file	Writes via buffer
Slower	Faster
No buffering	Uses buffering
Suitable for small data	Suitable for large data
Simple writing	Efficient writing

★ Difference Between FileReader and BufferedReader

FileReader

Definition:

FileReader is a **character stream class** used to **read data directly from a file character by character**.

♦ Key Points

- Reads **one character at a time**
- Directly accesses the file
- **Slower** for large files
- Simple to use
- Part of java.io package

♦ Example

```
FileReader fr = new FileReader("data.txt");
int ch;
while((ch = fr.read()) != -1) {
    System.out.print((char) ch);
}
fr.close();
```

BufferedReader

Definition:

BufferedReader is a character stream class that **reads data using a buffer**, which improves reading performance.

♦ Key Points

- Reads data **in chunks (buffered)**
- Uses FileReader internally
- **Faster** than FileReader
- Can read **line by line** using readLine()
- Suitable for large files

♦ Example

```
BufferedReader br = new BufferedReader(new FileReader("data.txt"));
String line;
while((line = br.readLine()) != null) {
    System.out.println(line);
}
br.close();
```

★ Main Difference Table (Very Important for Exam)

FileReader			BufferedReader	
Reads character by character			Reads using buffer	
Direct file access			Uses memory	buffer
Slower			Faster	
No readLine() method			Supports readLine()	
Best for small files			Best for large files	

★ FileInputStream and FileOutputStream in Java

📘 FileInputStream

Definition

FileInputStream is a **byte stream class** used to **read binary data** (bytes) from a file.

📌 It is mainly used for:

- Reading **images**
- Reading **audio/video**
- Reading **binary files**
- Reading **any type of file as bytes**

📦 Package:

java.io

✅ Simple Example: Reading a File

```
import java.io.FileInputStream;
public class FileInputDemo {
    public static void main(String[] args) throws Exception {

        FileInputStream fis = new FileInputStream("data.txt");
        int ch;

        while ((ch = fis.read()) != -1) {
            System.out.print((char) ch);
        }

        fis.close();
    }
}
```

}

Explanation

- read() reads one byte at a time
- Returns -1 at end of file
- Data is read in binary form

Explanation

- read() reads **one byte at a time**
- Returns -1 at **end of file**
- Data is read in **binary form**

 It is mainly used for:

- Writing **images**
- Writing **audio/video**
- Writing **binary files**
- Writing **raw byte data**

 Package:

java.io

Simple Example: Writing to a File

```
import java.io.FileOutputStream;

public class FileOutputDemo {
    public static void main(String[] args) throws Exception {

        FileOutputStream fos = new FileOutputStream("data.txt");
        String msg = "Welcome to Java File Handling";

        byte[] b = msg.getBytes();
        fos.write(b);

        fos.close();
        System.out.println("Data written successfully");
    }
}
```

Explanation

- Data is converted into **bytes**
- write() writes bytes to file
- File is created if not exists

★ Difference Between FileInputStream and FileOutputStream

FileInputStream	FileOutputStream
Reads data from file	Writes data to file
Input stream	Output stream
Used to read bytes	Used to write bytes
read() method	write() method
File → Program	Program → File
Used for reading images, audio	Used for writing images, audio

★ Introduction to Applet

📘 What is an Applet?

An **Applet** is a **small Java program** that runs **inside a web browser** or an **applet viewer**. Applet programs are mainly used to create **interactive web applications**.

📦 Applet Package

Applet classes are available in:

```
java.applet
```

★ Key Points about Applet

- Applet is **not a standalone program**
- It runs inside a **browser or appletviewer**
- It does **not contain main() method**
- It uses **life-cycle methods**
- Applet is **platform independent**
- Applet works in **client-side environment**

★ Applet Life Cycle

An applet passes through the following stages:

1. `init()` – called once when applet starts
2. `start()` – called when applet becomes active
3. `paint()` – used to draw output
4. `stop()` – called when applet is stopped
5. `destroy()` – called when applet is removed

How to run applet ?

To run a Java applet, you typically use either an applet viewer or a web browser that supports Java. Let's go step by step.

Write the Applet Code: (all code place in this path only)" C:\Program Files\Java\jdk-1.8\bin", inside bin folder only Appletviewer work.

If you get this error : MyApplet.java:4: error: error while writing MyApplet: MyApplet.class (Access is denied) 🙄

Check file permissions: Make sure the directory where you're saving MyApplet.java allows write access. If you're on Windows:

- Right-click the folder where the file is.
- Go to Properties > Security.
- Ensure your user account has Write permissions.
- And always Run CMD in Run as Administrator.

Create a simple Java applet program — MyApplet.java:

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

public class MyApplet extends Applet {
    @Override
    public void paint(Graphics g) {
        g.drawString("Welcome to Java Applet!", 50, 50);
    }
}
```

Create .html file for applet view

```
<applet code="MyApplet.class" width="300" height="150"></applet>
```

Compile the Applet:

Open your terminal (or command prompt) and navigate to the directory where your .java file is located. Then compile the applet using:

```
javac MyApplet.java
```

Run the Applet:

Option 1: Using the applet viewer (Recommended)

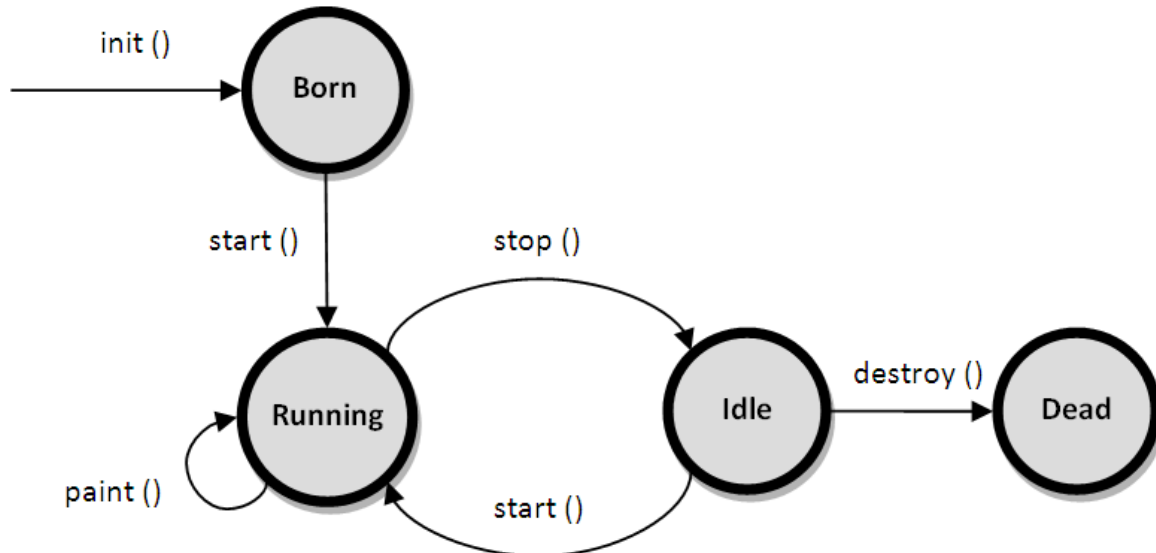
Java provides the appletviewer tool to run applets directly without a browser. In the terminal, type:

```
appletviewer MyApplet.html
```

Appletviewer only work in JDK 8 or lower version

You can download from here [jdk 8](#)

Lifecycle of an Applet:



```
public void init()
```

This method is called exactly once in an applet's life, when the applet is first loaded. It's normally used to read PARAM tags, start downloading any other images or media files you need, and to set up the user interface. Most applets have init() methods.

<param> tag is used to pass parameters from an HTML file to the applet.

Here's a simple example of an applet with the public void init() method:

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

public class SimpleApplet extends Applet {

    // init() method - called when the applet is first loaded
    public void init() {
        setBackground(java.awt.Color.lightGray); // Setting background color
    }

    // paint() method - called to draw the content of the applet
    public void paint(Graphics g) {
        g.drawString("Hello, this is a simple applet!", 20, 50);
    }
}
```

```
<html>
  <body>
    <applet code="SimpleApplet.class" width="300" height="150"></applet>
  </body>
</html>
```

add <PARAM> tags to pass values to an applet. Here's a simple example:

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

public class ParamApplet extends Applet {
    String message;

    // init() method to read parameter from HTML
    public void init() {
        message = getParameter("greeting"); // Reading the PARAM value
        if (message == null) {
            message = "Hello, default message!";
        }
    }

    // paint() method to display the message
    public void paint(Graphics g) {
        g.drawString(message, 20, 50);
    }
}

<html>
  <body>
    <applet code="ParamApplet.class" width="300" height="150">
      <param name="greeting" value="Welcome to the Applet World!" />
    </applet>
  </body>
</html>
```

public void start ()

This method is called at least once in an applet's life, when the applet is started or restarted. In some cases it may be called more than once. Many applets you write will not have explicit start)

method and will merely inherit one from their super class. A start() method is often used to start any threads the Applet will need while it runs.

Here's a simple example of an applet using the public void start() method:

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

public class StartMethodApplet extends Applet {
    String message;

    // init() method - called once when the applet is first loaded
    public void init() {
        message = "Applet initialized!";
    }

    // start() method - called every time the applet is started or revisited
    public void start() {
        message = "Applet started!";
        repaint(); // Requests the applet to redraw
    }

    // paint() method - called to display the message
    public void paint(Graphics g) {
        g.drawString(message, 20, 50);
    }
}
```

public void stop():

The public void stop() method in an applet is a lifecycle method that's called when the applet becomes inactive — like when the user navigates away from the page, minimizes the browser, or the applet is no longer visible.

Key points about stop():

It's automatically called by the browser or applet viewer.

It's used to pause activities like animations, background threads, or music.

It's optional — you only need it if you have processes that should pause when the applet isn't active.

Simple example:

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

public class StopExampleApplet extends Applet {
    String message;

    public void start() {
        message = "Applet started!";
        repaint();
    }

    public void stop() {
        message = "Applet stopped!";
        repaint();
    }

    public void paint(Graphics g) {
```

```
        g.drawString(message, 20, 50);
    }
}
```

`public void destroy():`

The `public void destroy()` method in an applet is the last lifecycle method — it's called when the applet is completely removed from memory. This usually happens when:

- The browser is closed.
- You navigate away from the page and the applet is no longer needed.
- The applet viewer decides to free resources.

Purpose of `destroy()`:

- Clean up resources (like closing files, network connections, or database connections).
- Stop background threads or timers.
- Release memory to prevent leaks.

Simple example:

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

public class DestroyExampleApplet extends Applet {
    String message;

    public void init() {
        message = "Applet initialized!";
    }

    public void start() {
        message = "Applet started!";
        repaint();
    }

    public void stop() {
        message = "Applet stopped!";
        repaint();
    }

    public void destroy() {
        System.out.println("Applet destroyed! Cleaning up resources...");
    }

    public void paint(Graphics g) {
        g.drawString(message, 20, 50);
    }
}
```

★ Playing Audio in Java Applet

■ Concept

In Java Applet, audio files (like .wav, .au) were played using the `AudioClip` interface from the `java.applet` package.

👉 Audio plays inside the applet environment.

📦 Package Used

java.applet

★ Important Methods of AudioClip

Method	Use
play()	Plays audio once
loop()	Plays audio continuously
stop()	Stops audio

★ Classic Java Applet Audio Example

⚠️ This code is for understanding and exam purpose only

Applets are deprecated after Java 9.

```
import java.applet.Applet;
import java.applet.AudioClip;
import java.net.URL;

public class AudioApplet extends Applet {

    AudioClip sound;

    public void init() {
        try {
            URL url = getCodeBase();
            sound = getAudioClip(url, "music.wav");
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    public void start() {
        sound.play();
    }

    public void stop() {
        sound.stop();
    }
}
```

★ HTML File

```
<applet code="AudioApplet.class" width="300" height="200">
</applet>
```

★ Graphics Class in Java

📖 What is Graphics Class?

The Graphics class is used to draw text, lines, shapes, and images on a component such as an Applet, Frame, or Panel.

📦 Package:

java.awt

★ Where is Graphics Used?

- Applet (paint(Graphics g))
- AWT (Frame, Panel)
- Swing (paintComponent(Graphics g))

It provides the drawing surface.

★ How Graphics Object is Obtained

In Swing

```
protected void paintComponent (Graphics g) {
    super.paintComponent (g);
}
```

★ Coordinate System (VERY IMPORTANT)

- Origin (0,0) is at top-left corner
 - x → right direction
 - y → downward direction

(0,0) —————▶ x

|

|

▼ y

★ Graphics Class Example (WITHOUT Applet)

✅ Example 1: Graphics using AWT

```
import java.awt.*;

public class GraphicsAWTExample extends Frame {
```

```

public GraphicsAWTExample() {
    setTitle("Graphics AWT Example");
    setSize(400, 300);
    setVisible(true);
}

@Override
public void paint(Graphics g) {
    g.drawString("Graphics Class Example", 50, 50);
    g.drawLine(50, 70, 250, 70);
    g.drawRect(50, 90, 120, 60);
    g.drawOval(200, 90, 100, 60);
    g.fillRect(50, 170, 120, 50);
}

public static void main(String[] args) {
    new GraphicsAWTExample();
}
}

```

✓ Example 2: Graphics using Swing

```

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

class MyPanel extends JPanel {

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        g.drawString("Graphics using Swing", 50, 50);
        g.setColor(Color.BLUE);
        g.drawRect(50, 80, 120, 60);

        g.setColor(Color.RED);
        g.fillOval(200, 80, 100, 60);
    }
}

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

        JFrame frame = new JFrame("Graphics Swing Example");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.add(new MyPanel());
        frame.setVisible(true);
    }
}

```

UNIT 4

★ Introduction to AWT and Swing

1] What is GUI in Java?

GUI (Graphical User Interface) allows users to interact with programs using windows, buttons, labels, text fields, menus, etc.

Java provides GUI support mainly through AWT and Swing.

1] What is GUI in Java?

GUI (Graphical User Interface) allows users to interact with programs using windows, buttons, labels, text fields, menus, etc.

Java provides GUI support mainly through AWT and Swing.

📦 Package:

Java.awt;

★ Key Characteristics of AWT1

- Uses native OS components
- Components are heavyweight
- Platform dependent look and feel
- Faster because it uses OS resources
- Limited set of GUI components

★ Common AWT Components

- Frame – main window
- Button – clickable button
- Label – displays text
- TextField – single-line input
- TextArea – multi-line input
- Checkbox – selection control

★ Simple AWT Example

```
import java.awt.*;
public class AWTDemo {
    public static void main(String[] args) {
        Frame f = new Frame("AWT Example");

        Label l = new Label("AWT is Working");
        Button b = new Button("Click");
        f.setLayout(new FlowLayout());
```

```

        f.add(l);
        f.add(b);

        f.setSize(300, 200);
        f.setVisible(true);
    }
}

```

3 Introduction to Swing

Definition

Swing is an advanced GUI toolkit built on top of AWT.

It provides lightweight, platform-independent components with a richer look and feel.

Package:

javax.swing

★ Key Characteristics of Swing

- Components are lightweight
- Platform independent
- Rich set of GUI controls
- Supports pluggable look and feel
- More flexible and powerful than AWT

★ Common Swing Components

- JFrame – main window
- JButton – button
- JLabel – text display
- JTextField – text input
- JTextArea – multi-line text
- JCheckBox – checkbox

★ Simple Swing Example

```

import javax.swing.*;

public class SwingDemo {
    public static void main(String[] args) {
        JFrame f = new JFrame("Swing Example");

        JLabel l = new JLabel("Swing is Recommended");
        JButton b = new JButton("Click");

        f.setLayout(new java.awt.FlowLayout());
        f.add(l);
        f.add(b);

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

```

AWT vs Swing

AWT	Swing
Heavyweight components	Lightweight components
Uses OS GUI	Uses Java GUI
Platform dependent	Platform independent
Limited controls	Rich controls
Older	Newer & advanced

Advantages and Limitations

✓ Advantages of AWT

- Simple
- Faster execution
- Direct OS support

✗ Limitations of AWT

- Platform dependent
- Limited components
- Less flexible UI

✓ Advantages of Swing

- Platform independent
- Rich components
- Customizable look and feel

✗ Limitations of Swing

- Slightly slower than AWT
- More complex than AWT

★ Difference Between AWT and Swing

📖 Overview

AWT and Swing are Java GUI toolkits used to create window-based desktop applications.

AWT is the older toolkit, while Swing is newer, more powerful, and preferred.

Basis	AWT (Abstract Window Toolkit)	Swing
Package	java.awt	javax.swing
Component Type	Heavyweight	Lightweight
OS Dependency	Platform dependent	Platform independent
Look and Feel	Native OS look	Pluggable look & feel
Components	Limited	Rich set of components
Performance	Faster (uses OS)	Slightly slower
Flexibility	Less flexible	Highly flexible
MVC Support	No	Yes
Usage Today	Basic / learning	Preferred / modern

Swing Components:

★ JFrame in Swing

📖 What is JFrame?

JFrame is a top-level container in Swing used to create a main window for a desktop application.

It can contain buttons, labels, text fields, menus, and other Swing components.

📦 Package:

javax.swing.JFrame

★ Key Points about JFrame

- Used to create the main application window
- Part of Swing (lightweight, platform-independent)
- Supports title bar, close/minimize buttons
- Can hold multiple components
- Replaces AWT's Frame

★ Basic Structure of a JFrame Program

```
import javax.swing.*;

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

        JFrame frame = new JFrame("My First JFrame");

        frame.setSize(400, 300);           // window size
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);           // show window
    }
}
```

★ Important Methods of JFrame.

Method	Purpose
setTitle(String)	Sets window title
setSize(w, h)	Sets window size
setLayout(LayoutManager)	Sets layout

add(Component)	Adds component
setVisible(true)	Displays window
setDefaultCloseOperation()	Action on close

Common close options:

- JFrame.EXIT_ON_CLOSE
- JFrame.HIDE_ON_CLOSE
- JFrame.DISPOSE_ON_CLOSE

They decide what happens when the user clicks the Close (X) button.

1) JFrame.EXIT_ON_CLOSE

♦ Meaning

- Closes the window
- Terminates the entire application
- JVM stops completely

✓ When to Use

- Single-window applications
- Beginner programs
- Most practical exams

📌 Most commonly used

2) JFrame.DISPOSE_ON_CLOSE

♦ Meaning

- Closes only the current window
- Releases memory/resources
- Application may continue running if other windows exist

✓ When to Use

- Multi-window applications
- Dialog boxes
- Child windows

3) JFrame.HIDE_ON_CLOSE

◆ Meaning

- Window becomes invisible
- Window object still exists in memory
- Can be shown again using setVisible(true)

✓ When to Use

- Temporarily hiding windows
- Reusable windows

4 JFrame.DO_NOTHING_ON_CLOSE

◆ Meaning

- Clicking close button does nothing
- Programmer must handle closing manually

✓ When to Use

- Show confirmation dialog
- Prevent accidental closing

★ Example: JFrame.HIDE_ON_CLOSE with Show Again

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

public class HideShowFrame {

    public static void main(String[] args) {

        // Main frame
        JFrame frame = new JFrame("HIDE_ON_CLOSE Example");
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());

        // Set HIDE_ON_CLOSE
        frame.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);

        JLabel label = new JLabel("Close me, I will hide!");
        JButton hideButton = new JButton("Hide Window");

        frame.add(label);
        frame.add(hideButton);

        frame.setVisible(true);

        // Second frame (controller)
        JFrame controlFrame = new JFrame("Show Window Again");
        controlFrame.setSize(300, 150);
        controlFrame.setLayout(new FlowLayout());

        JButton showButton = new JButton("Show Hidden Window");
        controlFrame.add(showButton);
```

```

controlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
controlFrame.setVisible(true);

// Hide action
hideButton.addActionListener(e -> frame.setVisible(false));

// Show again action
showButton.addActionListener(e -> frame.setVisible(true));
}
}

```

★ Complete Comparison Table

Close Option	Window Closed	App Ends	Memory Released	Typical Use
EXIT_ON_CLOSE	Yes	Yes	Yes	Simple apps
DISPOSE_ON_CLOSE	Yes	No	Yes	Multi-window
HIDE_ON_CLOSE	No (hidden)	No	No	Reusable UI
DO_NOTHING_ON_CLOSE	No	No	No	Confirmation

★ JPanel in Swing

📦 What is JPanel?

JPanel is a **lightweight container** in Swing used to **group and organize components** such as buttons, labels, text fields, etc., inside a window (JFrame).

It helps in **layout management**, **component grouping**, and **custom drawing**.

📦 Package:

javax.swing.JPanel

★ Key Features of JPanel

- Acts as a **sub-container** inside JFrame
- Used to **organize components** logically
- Supports **layout managers** (FlowLayout, BorderLayout, GridLayout, etc.)
- Can be used for **custom drawing** using paintComponent()
- Lightweight and platform independent

★ Why Use JPanel Instead of Adding Directly to JFrame?

- ✓ Better organization
- ✓ Multiple layouts in one window
- ✓ Easier design for complex GUIs

✓ Required for custom painting

★ Basic Structure Using JPanel

```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.add(new JButton("Button"));
frame.add(panel);
```

✓ Simple Working Example: JPanel with Components

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

public class JPanelDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JPanel Example");
        JPanel panel = new JPanel();           // Create panel
        panel.setLayout(new FlowLayout());     // Set layout
        JLabel label = new JLabel("Inside JPanel");
        JButton button = new JButton("Click Me");
        panel.add(label);                      // Add components to panel
        panel.add(button);
        frame.add(panel);                      // Add panel to frame
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

★ Using Multiple JPanels (Layout Management)

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

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

        JFrame frame = new JFrame("Multiple JPanel Example");

        JPanel topPanel = new JPanel();
        topPanel.add(new JLabel("Top Panel"));

        JPanel bottomPanel = new JPanel();
        bottomPanel.add(new JButton("OK"));

        frame.setLayout(new BorderLayout());
        frame.add(topPanel, BorderLayout.NORTH);
        frame.add(bottomPanel, BorderLayout.SOUTH);

        frame.setSize(300, 200);
    }
}
```

```

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

```

```

import java.awt.*;
panel.setBackground(Color.CYAN); // Set background color

```

Key Methods of JPanel:

Method	Description
setBackground(Color c)	Sets background color
add(Component c)	Adds a component to the panel
setLayout(LayoutManager mgr)	Changes layout manager
setSize(int width, int height)	Sets panel size

★ JLabel in Swing

📖 What is JLabel?

JLabel is a Swing component used to **display text, images (icons), or both** in a GUI. It is **not editable** and is mainly used for **labels, headings, and messages**.

📦 Package:

javax.swing.JLabel

★ Key Features of JLabel

- Displays **text, icon (image), or both**
- **Non-editable** (user cannot change text)
- Supports **alignment** (left, center, right)
- Can use **fonts, colors, and borders**
- Lightweight and platform independent

★ Why Use JLabel?

- ✓ To show titles, instructions, messages
- ✓ To label text fields (Name:, Age:)
- ✓ To display images/icons

★ Basic Syntax

JLabel label = new JLabel("Hello World");

✓ Simple Working Example: JLabel with Text

```
import javax.swing.*;

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

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

        JLabel label = new JLabel("Welcome to Swing");
        frame.add(label);

        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

✓ JLabel with Font, Color & Alignment

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

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

        JFrame frame = new JFrame("Styled JLabel");

        JLabel label = new JLabel("Styled Label", JLabel.CENTER);
        label.setFont(new Font("Arial", Font.BOLD, 16));
        label.setForeground(Color.BLUE);

        frame.add(label);
        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Key Methods of JLabel

Method	Description
setText(String text)	Sets the text of the label

setIcon(Icon icon)	Sets an image icon
setFont(Font font)	Sets the font of the text
setForeground(Color c)	Sets the text color
setHorizontalAlignment(int alignment)	Aligns text horizontally (LEFT, CENTER, RIGHT)
setVerticalAlignment(int alignment)	Aligns text vertically (TOP, CENTER, BOTTOM)
setHorizontalTextPosition(int position)	Positions text relative to an image (LEFT, RIGHT, etc.)
setVerticalTextPosition(int position)	Positions text relative to an image (TOP, BOTTOM, etc.)

★ Important Methods of JLabel

JButton is a Swing component used to create a clickable button in a GUI application. It can trigger actions when clicked, making it an essential part of interactive applications.

Creating a Basic JButton

Here's a simple example of adding a button to a JFrame:

```
import javax.swing.*;

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

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

        // Add button to frame
        frame.add(button);

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

Explanation:

- new JButton("Click Me") → Creates a button with the label "Click Me".
- frame.add(button) → Adds the button to the frame.
- setVisible(true) → Displays the frame.

Key Methods of JButton:

Method	Description
setText(String text)	Sets the button's label
setIcon(Icon icon)	Adds an image to the button

setEnabled(boolean enabled)	Enables or disables the button
setFont(Font font)	Changes the button's font
setForeground(Color color)	Sets the text color
setBackground(Color color)	Sets the background color
addActionListener(ActionListener l)	Adds a click event listener

JRadioButton in Swing

JRadioButton is a Swing component used for creating radio buttons, which allow users to select one option from a group. When multiple JRadioButtons are grouped, selecting one automatically deselects the others.

Basic Example of JRadioButton

Here's how you can create a single radio button in a JFrame:

```
import javax.swing.*;

public class RadioButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JRadioButton Example");

        // Create a JRadioButton
        JRadioButton radioButton = new JRadioButton("Option 1");

        // Add radio button to frame
        frame.add(radioButton);

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

Grouping JRadioButton (Selecting Only One Option)

To ensure that only one option is selected at a time, use ButtonGroup:

```
import javax.swing.*;

public class RadioButtonGroup {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JRadioButton Group Example");

        // Create radio buttons
        JRadioButton option1 = new JRadioButton("Option 1");
        JRadioButton option2 = new JRadioButton("Option 2");
        JRadioButton option3 = new JRadioButton("Option 3");

        // Group the radio buttons
        ButtonGroup group = new ButtonGroup();
        group.add(option1);
        group.add(option2);
        group.add(option3);

        // Create a panel and add buttons to it
        JPanel panel = new JPanel();
    }
}
```

```

        panel.add(option1);
        panel.add(option2);
        panel.add(option3);

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

```

Explanation:

- Create multiple JRadioButtons.
- Use ButtonGroup to group them together (ensures only one can be selected).
- Add them to a JPanel before adding to the frame.

Key Methods of JRadioButton

Method	Description
setText(String text)	Sets the radio button's label
setSelected(boolean selected)	Selects/deselects the button
isSelected()	Checks if the button is selected
setEnabled(boolean enabled)	Enables/disables the button

JCheckBox in Swing

JCheckBox is a Swing component used to toggle selections ON/OFF. Unlike JRadioButton, multiple checkboxes can be selected at the same time.

Basic Example of JCheckBox

Here's how you can create a single checkbox in a JFrame:

```

import javax.swing.*;

public class CheckBoxExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JCheckBox Example");

        // Create a JCheckBox
        JCheckBox checkBox = new JCheckBox("Accept Terms and Conditions");

        // Add checkbox to frame
        frame.add(checkBox);

        // Set frame properties
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

```

Key Methods of JCheckBox

Method	Description
--------	-------------

setText(String text)	Sets the checkbox label
setSelected(boolean selected)	Selects/deselects the checkbox
isSelected()	Checks if the checkbox is selected
setEnabled(boolean enabled)	Enables/disables the checkbox
setFont(Font font)	Changes the text font
setForeground(Color color)	Sets text color
setBackground(Color color)	Sets background color

JTextField in Swing

JTextField is a Swing component used to take single-line text input from the user. It is commonly used in forms, search bars, and user input fields.

Basic Example of JTextField

Here's how you can create a simple text field in a JFrame:

```
import javax.swing.*;

public class TextFieldExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTextField Example");

        // Create a JTextField
        JTextField textField = new JTextField(20); // 20 columns width

        // Add text field to frame
        frame.add(textField);

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

Getting Input from JTextField

To retrieve the text entered in a JTextField, use the `getText()` method:

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

public class TextFieldInput {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTextField Input Example");

        // Create a text field and a button
        JTextField textField = new JTextField(20);
        JButton button = new JButton("Submit");
        JLabel label = new JLabel("Enter something:");

        // Add ActionListener to the button
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String inputText = textField.getText();
                JOptionPane.showMessageDialog(frame, "You entered: " + inputText);
            }
        });
    }
}
```

```

});
// Create panel and add components
JPanel panel = new JPanel();
panel.add(label);
panel.add(textField);
panel.add(button);

frame.add(panel);
frame.setSize(350, 150);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
}

```

Setting Default Text in JTextField

You can set an initial value using `setText()`:

```
textField.setText("Enter your name...");
```

Clearing JTextField After Input

To clear the text field after submitting:

```

button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        textField.setText(""); // Clears the text field
    }
});

```

Disabling and Enabling JTextField

You can disable/enable a text field using:

```
textField.setEnabled(false); // Disables the text field
```

```
textField.setEnabled(true); // Enables the text field
```

Key Methods of JTextField

Method	Description
setText(String text)	Sets default text in the field
getText()	Retrieves user input
setEnabled(boolean enabled)	Enables/disables the text field
setFont(Font font)	Changes the text font
setForeground(Color color)	Sets text color
setBackground(Color color)	Sets background color
setHorizontalAlignment(int alignment)	Aligns text (LEFT, CENTER, RIGHT)

JPasswordField in Swing

JPasswordField is a Swing component used for password input, similar to JTextField, but it hides the characters entered by default (e.g., ****).

Basic Example of JPasswordField

Here's how you can create a simple password field in a JFrame:

```
import javax.swing.*;

public class PasswordFieldExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JPasswordField Example");

        // Create a password field
        JPasswordField passwordField = new JPasswordField(20); // 20 columns width

        // Add password field to frame
        frame.add(passwordField);

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

Getting Password Input (getPassword())

Unlike JTextField which uses getText(), JPasswordField uses getPassword(), which returns a char[] instead of String.

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

public class PasswordInput {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JPasswordField Input Example");

        // Create password field and button
        JPasswordField passwordField = new JPasswordField(20);
        JButton button = new JButton("Submit");
        JLabel label = new JLabel("Enter your password:");

        // ActionListener for the button
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                char[] password = passwordField.getPassword();
                String passText = new String(password); // Convert char[] to String
                JOptionPane.showMessageDialog(frame, "Your password is: " + passText);
            }
        });

        // Create panel and add components
        JPanel panel = new JPanel();
        panel.add(label);
        panel.add(passwordField);
        panel.add(button);
    }
}
```

```

        frame.add(panel);
        frame.setSize(350, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

```

Enabling/Disabling JPasswordField

`passwordField.setEnabled(false);` // Disables the password field

`passwordField.setEnabled(true);` // Enables the password field

Key Methods of JPasswordField

Method Description

getPassword()	Returns the password as a char[] array
setEchoChar(char c)	Sets the character used to hide password (*, #, etc.)
setText(String text)	Sets a default password
setEnabled(boolean enabled)	Enables/disables the password field
setFont(Font font)	Changes the text font
setForeground(Color color)	Sets text color
setBackground(Color color)	Sets background color

JTextArea in Swing

JTextArea is a Swing component used for multi-line text input. Unlike JTextField, which only supports single-line input, JTextArea allows users to enter and edit multiple lines of text.

Basic Example of JTextArea

Here's a simple example of creating a JTextArea inside a JFrame:

```

import javax.swing.*;

public class K_TextAreaExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTextArea Example");
        JPanel panel = new JPanel();

        // Create a JTextArea with 5 rows and 20 columns
        JTextArea textArea = new JTextArea(5, 20);

        // Add the text area to the frame
        panel.add(textArea);
        frame.add(panel);

        // Set frame properties
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

```

Key Methods of JTextArea

Method	Description
setText(String text)	Sets default text
getText()	Retrieves user input
append(String text)	Adds text without removing existing text
setEditable(boolean editable)	Enables/disables editing
setForeground(Color color)	Sets text color
setBackground(Color color)	Sets background color
setLineWrap(true)	Enables text wrapping
setWrapStyleWord(true)	Wraps text at word boundaries

JScrollBar in Swing

JScrollBar is a Swing component used to create a scrollable view. It can be vertical or horizontal and is often used in JScrollPane, JTextArea, JPanel, and other large UI elements.

Basic Example of JScrollBar

Here's how you can create a simple vertical scrollbar in a JFrame:

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

public class ScrollBarExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JScrollBar Example");

        // Create a vertical scrollbar
        JScrollBar scrollbar = new JScrollBar(JScrollBar.VERTICAL, 30, 20, 0, 100);

        // Add AdjustmentListener to track changes
        scrollbar.addAdjustmentListener(new AdjustmentListener() {
            public void adjustmentValueChanged(AdjustmentEvent e) {
                System.out.println("Scrollbar Value: " + scrollbar.getValue());
            }
        });

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

Parameter	Description
JScrollBar.VERTICAL	Sets orientation (VERTICAL or HORIZONTAL)
30	Initial scroll position (default value)
20	Scroll thumb size (visible area)
0	Minimum value
100	Maximum value

Key Methods of JScrollBar

Method	Description
setValue(int)	Sets scrollbar position
getValue()	Gets the current scrollbar position
setMinimum(int)	Sets minimum scroll value
setMaximum(int)	Sets maximum scroll value
setOrientation(JScrollBar.HORIZONTAL)	Changes orientation
setUnitIncrement(int)	Adjusts small scrolling steps
setBlockIncrement(int)	Adjusts large scrolling steps

Example:

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

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

        JFrame frame = new JFrame("JScrollBar Methods Demo");
        frame.setSize(400, 200);
        frame.setLayout(new FlowLayout());
        JLabel label = new JLabel("Value: 0");
        // Create scrollbar
        JScrollBar scrollbar = new JScrollBar(JScrollBar.VERTICAL, 30, 20, 0, 100);
        // Apply methods
        scrollbar.setMinimum(0); // minimum value
        scrollbar.setMaximum(101); // maximum value
        scrollbar.setValue(30); // initial position
        scrollbar.setOrientation(JScrollBar.HORIZONTAL); // horizontal bar
        scrollbar.setUnitIncrement(5); // small step
        scrollbar.setBlockIncrement(20); // large step
        // Display current value
        scrollbar.addAdjustmentListener(new AdjustmentListener() {
            public void adjustmentValueChanged(AdjustmentEvent e) {
                int val = scrollbar.getValue(); // getValue()
                label.setText("Value: " + val);
            }
        });
        frame.add(label);
        frame.add(scrollbar);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

JComboBox in Swing

JComboBox is a drop-down menu that allows users to select one option from a list. It is commonly used for choosing items like countries, colors, languages, etc.

Basic Example of JComboBox

Here's how you can create a simple JComboBox in a JFrame:

```
import javax.swing.*;

public class ComboBoxExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JComboBox Example");

        // Create an array of items
        String[] languages = { "Java", "Python", "C++", "JavaScript" };

        // Create a JComboBox with the array
        JComboBox comboBox = new JComboBox(languages);

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

Handling Events: Get Selected Item

To detect changes in the selected item, use an ActionListener:

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

```
// Add ActionListener to detect selection changes
comboBox.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String selected = (String) comboBox.getSelectedItem();
        JOptionPane.showMessageDialog(frame, "You selected: " + selected);
    }
});
```

Adding and Removing Items Dynamically

You can add or remove items in the dropdown at runtime:

```
comboBox.addItem("Swift"); // Adds a new item
comboBox.removeItem("C++"); // Removes an item
comboBox.removeAllItems(); // Clears all items
```

Setting Default Selected Item

You can pre-select an item before displaying the GUI:

```
comboBox.setSelectedItem("Python"); // Selects "Python" by default
```

Or, if you know the index of the item:

```
comboBox.setSelectedIndex(1); // Selects the second item (index starts at 0)
```

Customizing JComboBox

Setting Font & Color

```
comboBox.setFont(new Font("Arial", Font.BOLD, 14));
```

```
comboBox.setForeground(Color.BLUE);
```

```
comboBox.setBackground(Color.LIGHT_GRAY);
```

Disabling Editing

By default, JComboBox allows selecting items but not typing. To make it editable:

```
comboBox.setEditable(true);
```

Key Methods of JComboBox

Method	Description
getSelectedItem()	Gets the selected item
setSelectedItem(Object item)	Sets the selected item
setSelectedIndex(int index)	Selects item by index
addItem(Object item)	Adds an item dynamically
removeItem(Object item)	Removes a specific item
removeAllItems()	Removes all items
setEditable(boolean editable)	Enables or disables text editing
setEnabled(boolean enabled)	Enables or disables selection

Example:

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

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

        JFrame frame = new JFrame("JComboBox Methods Demo");
        frame.setSize(400, 300);
        frame.setLayout(null);
        // TextField
        JTextField textField = new JTextField();
        textField.setBounds(120, 2, 200, 25);

        JLabel label = new JLabel("Selected: ");
        label.setBounds(50, 20, 300, 25);

        // Create JComboBox
        JComboBox<String> combo = new JComboBox<>();
        combo.setBounds(50, 60, 200, 25);

        // Add initial items
```

```

combo.addItem("Java");
combo.addItem("C");
combo.addItem("C++");

// Buttons
JButton btnGet = new JButton("Get Selected");
btnGet.setBounds(50, 100, 140, 30);

JButton btnAdd = new JButton("Add Python");
btnAdd.setBounds(200, 100, 140, 30);

JButton btnRemove = new JButton("Remove C");
btnRemove.setBounds(50, 140, 140, 30);

JButton btnClear = new JButton("Clear All");
btnClear.setBounds(200, 140, 140, 30);

// Button
JButton btnadditem = new JButton("Add to ComboBox");
btnadditem.setBounds(50, 200, 160, 30);

// Actions
btnGet.addActionListener(e -> {
    Object item = combo.getSelectedItem(); // getItem()
    label.setText("Selected: " + item);
});

btnAdd.addActionListener(e -> {
    combo.addItem("Python"); // addItem()
});

btnRemove.addActionListener(e -> {
    combo.removeItem("C"); // removeItem()
});

btnClear.addActionListener(e -> {
    combo.removeAllItems(); // removeAllItems()
});

// Button Action
btnadditem.addActionListener(e -> {
    String text = textField.getText(); // get value from textfield

    if (!text.isEmpty()) {
        combo.addItem(text); // add to combobox
        textField.setText(""); // clear textfield
    } else {
        JOptionPane.showMessageDialog(frame, "Please enter some text");
    }
});

// Other method examples
combo.setSelectedIndex(0); // setSelectedIndex()
combo.setEditable(true); // setEditable()
combo.setEnabled(true); // setEnabled()

// Add components
frame.add(label);
frame.add(combo);
frame.add(btnGet);
frame.add(btnAdd);
frame.add(btnRemove);
frame.add(btnClear);

```

```

        frame.add(textField);
        frame.add(btnadditem);

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

```

JList in Swing

JList is a component in Swing that allows users to select one or multiple items from a list. It is commonly used for menus, selection lists, and navigation panels.

Basic Example of JList

Here's how you can create a simple JList in a JFrame:

```

import javax.swing.*;

public class JListExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JList Example");

        // Create an array of items
        String[] languages = { "Java", "Python", "C++", "JavaScript" };
        JList list = new JList();
        list.setListData(languages);

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

```

Handling Events: Get Selected Item

To detect when an item is selected, use a ListSelectionListener:

```

import javax.swing.*;
import javax.swing.event.*;

public class JListEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JList Event Example");
        String[] languages = { "Java", "Python", "C++", "JavaScript" };
        JList<String> list = new JList<>(languages);
        JScrollPane scrollPane = new JScrollPane(list);
        JLabel label = new JLabel("Selected: None");
    }
}

```

```

// Add event listener for selection changes
list.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        if (!e.getValueIsAdjusting()) { // Avoid double triggers
            String selected = list.getSelectedValue();
            label.setText("Selected: " + selected);
        }
    }
});
JPanel panel = new JPanel();
panel.add(scrollPane);
panel.add(label);

frame.add(panel);
frame.setSize(350, 200);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
}

```

Customizing JList Appearance

Change Font & Colors

```
list.setFont(new Font("Arial", Font.BOLD, 14));
```

```
list.setForeground(Color.BLUE);
```

```
list.setBackground(Color.LIGHT_GRAY);
```

Key Methods of JList

Method	Description
getSelectedValue()	Gets the selected item
getSelectedValuesList()	Gets multiple selected items
setSelectionMode(int mode)	Sets selection mode (SINGLE_SELECTION, MULTIPLE_INTERVAL_SELECTION)
setListData(Object[] items)	Sets new data in the list
setFont(Font font)	Sets font style
setForeground(Color color)	Sets text color
setBackground(Color color)	Sets background color

Example :

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.List;

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

```

```

JFrame frame = new JFrame("JList Methods Demo");
frame.setSize(400, 300);
frame.setLayout(null);

// Initial data
String[] languages = {"Java", "C", "C++", "Python", "PHP"};

// Create JList
JList<String> list = new JList<>(languages);
list.setBounds(50, 40, 150, 120);

// Apply methods
list.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION); //
multiple selection
list.setFont(new Font("Arial", Font.BOLD, 14)); //
font
list.setForeground(Color.BLUE); //
text color
list.setBackground(Color.LIGHT_GRAY); //
background

// Button to get selected values
JButton btnGet = new JButton("Get Selected");
btnGet.setBounds(220, 40, 130, 30);

// Button to change list data
JButton btnSetData = new JButton("Change Data");
btnSetData.setBounds(220, 80, 130, 30);

// Label to display result
JLabel label = new JLabel("Selected: ");
label.setBounds(50, 180, 300, 30);

// Get selected values
btnGet.addActionListener(e -> {
    List<String> selected = list.getSelectedValuesList(); //
getSelectedValuesList()
    label.setText("Selected: " + selected);
});

// Change list items dynamically
btnSetData.addActionListener(e -> {
    String[] newData = {"HTML", "CSS", "JavaScript"}; //
setListData()
    list.setListData(newData);
});

// Add components
frame.add(list);
frame.add(btnGet);
frame.add(btnSetData);
frame.add(label);

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

```

JToggleButton in Swing

JToggleButton is a button that works like an on/off switch (toggle behavior). When clicked, it stays pressed (selected) until clicked again. It is useful for settings, themes (dark/light mode), or switching modes in an application.

Basic Example of JToggleButton

Here's a simple example of JToggleButton in a JFrame:

```
import javax.swing.*;
public class ToggleButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JToggleButton Example");
        // Create a JToggleButton
        JToggleButton toggleButton = new JToggleButton("OFF");

        frame.add(toggleButton);
        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Customizing JToggleButton

You can change font, background, and text color:

```
toggleButton.setFont(new Font("Arial", Font.BOLD, 16));
toggleButton.setForeground(Color.WHITE);
toggleButton.setBackground(Color.DARK_GRAY);
```

```
// Using traditional ActionListener instead of lambda
toggleButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (toggleButton.isSelected()) {
            toggleButton.setText("ON");
        } else {
            toggleButton.setText("OFF");
        }
    }
});
```

Key Methods of JToggleButton

Method	Description
isSelected()	Returns true if button is pressed (ON)
setText(String text)	Changes the button text
setForeground(Color color)	Sets text color
setBackground(Color color)	Sets background color
setFont(Font font)	Sets font style

setEnabled(boolean enabled)	Enables/disables the button
------------------------------------	-----------------------------

Example :

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

public class ToggleButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JToggleButton Example");
        frame.setLayout(new FlowLayout());
        JLabel label = new JLabel("State: OFF");

        // Create a JToggleButton
        JToggleButton toggleButton = new JToggleButton("OFF");
        toggleButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (toggleButton.isSelected()) { // check ON/OFF state
                    toggleButton.setText("ON");
                    label.setText("State: ON");
                } else {
                    toggleButton.setText("OFF");
                    label.setText("State: OFF");
                }
            }
        });

        frame.add(toggleButton);
        frame.add(label);
        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

JTabbedPane in Swing

JTabbedPane is a Swing component that lets you create tabbed navigation, similar to browser tabs. It allows you to switch between multiple panels without opening new windows.

Basic Example of JTabbedPane

Here's a simple example where we create a JTabbedPane with three tabs:

```
import javax.swing.*;

public class R_TabbedPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTabbedPane Example");

        // Create a JTabbedPane
        JTabbedPane tabbedPane = new JTabbedPane();

        // Create panels for different tabs
        JPanel tab1 = new JPanel();
```

```

tab1.add(new JLabel("This is Tab 1"));

JPanel tab2 = new JPanel();
tab2.add(new JLabel("This is Tab 2"));

JPanel tab3 = new JPanel();
tab3.add(new JLabel("This is Tab 3"));

// Add tabs to JTabbedPane
tabbedPane.addTab("Tab 1", tab1);
tabbedPane.addTab("Tab 2", tab2);
tabbedPane.addTab("Tab 3", tab3);

// Add JTabbedPane to the frame
frame.add(tabbedPane);
frame.setSize(400, 200);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
}

```

Switching Tabs Programmatically

To set a specific tab as active:

```
tabbedPane.setSelectedIndex(1); // Opens Tab 2 (index starts from 0)
```

Removing or Renaming Tabs

To remove a tab:

```
tabbedPane.removeTabAt(1); // Removes the second tab
```

To rename a tab:

```
tabbedPane.setTitleAt(0, "New Name"); // Renames Tab 1
```

Handling Tab Change Events

You can detect when the user switches tabs:

```

// Handle tab change event using an anonymous class
tabbedPane.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        int selectedTab = tabbedPane.getSelectedIndex();
        System.out.println("Switched to Tab: " + (selectedTab + 1));
    }
});

```

Common JTabbedPane Methods

Method	Description
addTab(String title, Component component)	Adds a new tab
setSelectedIndex(int index)	Selects a tab by index
getSelectedIndex()	Gets the index of the selected tab

getTitleAt(int index)	Gets the title of a tab
removeTabAt(int index)	Removes a tab by index
removeAll()	Removes all tabs
setEnabledAt(int index, boolean enabled)	Enables/disables a tab
isEnabledAt(int index)	Checks if a tab is enabled
setTitleAt(int index, String title)	Changes the title of a tab
setIconAt(int index, Icon icon)	Sets an icon for a tab
setToolTipTextAt(int index, String text)	Sets a tooltip for a tab
setBackgroundAt(int index, Color color)	Changes tab background color
setForegroundAt(int index, Color color)	Changes tab text color
setTabPlacement(int placement)	Changes tab position (TOP, BOTTOM, LEFT, RIGHT)
addChangeListener(ChangeListener l)	Listens for tab change events

JSlider in Swing

JSlider is a Swing component that allows users to select a value from a range by moving a sliding knob. It is commonly used for volume control, brightness adjustment, or setting numeric values.

Basic Example of JSlider

Here's a simple example where we create a JSlider that ranges from 0 to 100 and displays the selected value.

```
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class JSliderExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JSlider Example");
        JPanel panel = new JPanel();
        // Create a JSlider (min: 0, max: 100, initial: 50)
        JSlider slider = new JSlider(0, 100, 50);
        // Display the selected value
        JLabel label = new JLabel("Value: 50");
        // Add change listener to update label when slider moves
        slider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                int value = slider.getValue();
                label.setText("Value: " + value);
            }
        });
        panel.add(slider);
        panel.add(label);
        frame.add(panel);
        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Commonly Used Methods of JSlider

Method	Description
new JSlider(min, max, value)	Creates a slider with a given range and initial value
setMinimum(int min)	Sets the minimum value of the slider
setMaximum(int max)	Sets the maximum value of the slider
setValue(int value)	Sets the current value of the slider
getValue()	Gets the current value of the slider
setMajorTickSpacing(int n)	Sets major tick spacing
setMinorTickSpacing(int n)	Sets minor tick spacing
setPaintTicks(boolean b)	Enables or disables tick marks
setPaintLabels(boolean b)	Enables or disables labels
setOrientation(int orientation)	Sets the slider orientation (JSlider.HORIZONTAL or JSlider.VERTICAL)

Example :

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

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

        JFrame frame = new JFrame("JSlider Methods Demo");
        frame.setSize(400, 250);
        frame.setLayout(null);
        JLabel label = new JLabel("Value: 50");
        label.setBounds(160, 30, 100, 25);
        // Create slider
        JSlider slider = new JSlider();
        // Apply methods
        slider.setMinimum(0);           // min
        slider.setMaximum(100);         // max
        slider.setValue(50);             // initial value
        slider.setMajorTickSpacing(20);  // major ticks
        slider.setMinorTickSpacing(5);  // minor ticks
        slider.setPaintTicks(true);      // show ticks
        slider.setPaintLabels(true);     // show labels
        slider.setOrientation(JSlider.HORIZONTAL); // orientation

        // Position slider (no layout)
        slider.setBounds(50, 80, 300, 70);

        // Listen to value changes
        slider.addChangeListener(e -> {
            int val = slider.getValue(); // getValue()
            label.setText("Value: " + val);
        });

        frame.add(label);
        frame.add(slider);

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

JProgressBar in Swing

JProgressBar is used to visually indicate the progress of a task (like file downloads, loading screens, or time-consuming operations).

Basic Example: Simple Progress Bar

```
import javax.swing.*;

public class T_JProgressBarExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JProgressBar Example");
        JPanel panel = new JPanel();

        // Create a progress bar (0 to 100, initial value 0)
        JProgressBar progressBar = new JProgressBar(0, 100);
        progressBar.setValue(50); // Set initial progress to 50%
        progressBar.setStringPainted(true); // Show percentage

        panel.add(progressBar);
        frame.add(panel);

        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Simulating Progress with a Timer

A progress bar is useful when updating in real-time. We can use a Swing Timer to increment the progress dynamically.

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

public class JProgressBarTimer {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JProgressBar with Timer");
        JPanel panel = new JPanel();

        JProgressBar progressBar = new JProgressBar(0, 100);
        progressBar.setStringPainted(true);

        // Timer to update progress every 100ms
        Timer timer = new Timer(100, new ActionListener() {
            int progress = 0;
            public void actionPerformed(ActionEvent e) {
                progress += 5;
                progressBar.setValue(progress);
                if (progress >= 100) {
                    ((Timer) e.getSource()).stop(); // Stop when complete
                }
            }
        });

        // Start the timer when the program runs
        timer.start();
    }
}
```

```

        panel.add(progressBar);
        frame.add(panel);

        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

```

Commonly Used Methods of JProgressBar

Method	Description
<code>new JProgressBar(min, max)</code>	Creates a progress bar with a range
<code>setValue(int value)</code>	Sets the current progress
<code>getValue()</code>	Gets the current progress value
<code>setStringPainted(boolean b)</code>	Shows the percentage text
<code>setIndeterminate(boolean b)</code>	Enables/disables indeterminate mode
<code>setForeground(Color c)</code>	Changes the progress bar color

Example :

```

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

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

        JFrame frame = new JFrame("JProgressBar Example");
        frame.setSize(400, 180);
        frame.setLayout(null);

        // Create progress bar with min & max
        JProgressBar bar = new JProgressBar(0, 100);

        bar.setValue(0); // initial value
        bar.setStringPainted(true); // show %
        bar.setForeground(Color.GREEN); // color
        //bar.setIndeterminate(true);

        bar.setBounds(50, 40, 300, 30);
        frame.add(bar);

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

        // Simulate progress
        for (int i = 0; i <= 100; i++) {
            bar.setValue(i); // setValue()
            try { Thread.sleep(40); } catch (Exception e) {}
        }

        // Uncomment to see indeterminate mode

    }
}

```

JTextPane in Swing

JTextPane is an advanced text **component** in **Swing** that supports rich text formatting, such as **bold**, *italic*, **colored** text, different fonts, and images. Unlike JTextArea, which only supports plain text, JTextPane is useful for styled documents.

Basic Example: Simple JTextPane

```
import javax.swing.*;

public class JTextPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTextPane Example");
        JPanel panel = new JPanel();

        // Create a JTextPane
        JTextPane textPane = new JTextPane();
        textPane.setText("Welcome to JTextPane!");

        // Add to a scroll pane
        JScrollPane scrollPane = new JScrollPane(textPane);
        scrollPane.setPreferredSize(new java.awt.Dimension(250, 100));

        panel.add(scrollPane);
        frame.add(panel);

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

Commonly Used Methods of JTextPane

Method	Description
setText(String text)	Sets the text in JTextPane
getText()	Gets the current text
setEditable(boolean b)	Enables/disables text editing
setCharacterAttributes(AttributeSet attr, boolean replace)	Applies styling to selected text
insertIcon(Icon icon)	Inserts an image into the text pane
setStyledDocument(StyledDocument doc)	Sets a custom document with styles

★ Menus in Swing (JMenuBar, JMenu, JMenuItem)

Swing provides menu components:

Component	Purpose
JMenuBar	Top menu bar (File, Edit, Help)

JMenu	Menu category (File, Edit)
JMenuItem	Actual command (Save, Exit)

JMenuBar, JMenu and JMenuItem are a part of Java Swing package. JMenuBar is an implementation of menu bar . the JMenuBar contains one or more JMenu objects, when the JMenu objects are selected they display a popup showing one or more JMenuItem . JMenu basically represents a menu . It contains several JMenuItem Object . It may also contain JMenu Objects (or submenu).

Constructors :

JMenuBar() : Creates a new MenuBar.

JMenu() : Creates a new Menu with no text.

JMenu(String name) : Creates a new Menu with a specified name.

JMenu(String name, boolean b) : Creates a new Menu with a specified name and boolean value specifies it as a tear-off menu or not. A tear-off menu can be opened and dragged away from its parent menu bar or menu.

Commonly used methods:

1. add(JMenu c) : Adds menu to the menu bar. Adds JMenu object to the Menu bar.
2. add(Component c) : Add component to the end of JMenu
3. add(Component c, int index) : Add component to the specified index of JMenu
4. add(JMenuItem menuItem) : Adds menu item to the end of the menu.
5. add(String s) : Creates a menu item with specified string and appends it to the end of menu.
6. getItem(int index) : Returns the specified menuItem at the given index

The JMenuItem class represents the actual item in a menu. All items in a menu should derive from class JMenuItem, or one of its subclasses. By default, it embodies a simple labeled menu item.

Sr.No	Constructor & Description
1	JMenuItem() Creates a JMenuItem with no set text or icon.

Example:

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

public class SimpleNotepad extends JFrame {

    JTextArea textArea;

    public SimpleNotepad() {
```

```

// Frame Title
setTitle("Notepad Application");
setSize(600, 400);

// Text Area
textArea = new JTextArea();
add(new JScrollPane(textArea));

// Create MenuBar
JMenuBar menuBar = new JMenuBar();

// Create Menu
JMenu fileMenu = new JMenu("File");

// Create Menu Items
JMenuItem newItem = new JMenuItem("New");
JMenuItem saveItem = new JMenuItem("Save");
JMenuItem exitItem = new JMenuItem("Exit");

// Add Items to Menu
fileMenu.add(newItem);
fileMenu.add(saveItem);
fileMenu.add(exitItem);

// Add Menu to MenuBar
menuBar.add(fileMenu);

// Set MenuBar in Frame
setJMenuBar(menuBar);

// -----
// Menu Item Actions
// -----

// New Option
newItem.addActionListener(e -> {
    textArea.setText("");
});

// Save Option
saveItem.addActionListener(e -> {
    try {
        FileWriter writer = new FileWriter("note.txt");
        writer.write(textArea.getText());
        writer.close();
        JOptionPane.showMessageDialog(this, "File Saved Successfully!");
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(this, "Error Saving File!");
    }
});

// Exit Option
exitItem.addActionListener(e -> {
    System.exit(0);
});

// Frame Settings
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true);
}

public static void main(String[] args) {

```

```
        new SimpleNotepad();  
    }  
}
```

LAYOUT MANAGERS:

In Java applets (and Java GUI programming in general), a layout manager is an object that arranges the components (like buttons, text fields, labels, etc.) in a container like an applet or a frame. It decides the position and size of each component based on rules defined by the specific layout manager you use.

The Applet class itself is a container, and it uses layout managers from the `java.awt` package to manage its components. By default, an applet uses `FlowLayout`, but you can change it to other layout managers like:

★ What is `FlowLayout` (Default Layout) in Java?

`FlowLayout` is a layout manager in Java that arranges components left to right in a row.

If there is no space left, it automatically moves components to the next line.

📦 Package:

java.awt.FlowLayout

FlowLayout (Default Layout):

- Arranges components in a row, left to right.
- Moves to the next line when out of space.
- Components stay in their preferred size.

★ Why it is called Default Layout?

FlowLayout is the default layout for:

- JPanel
- Panel

Content area of JFrame (when not changed manually)

So if you don't set any layout, FlowLayout is used automatically.

★ Features of FlowLayout

- Places components **left** → **right**
- Moves to next line when space is full
- Keeps component preferred size
- Supports alignment:
 - LEFT
 - CENTER (default)
 - RIGHT

★ Constructors

```
FlowLayout() // center alignment
FlowLayout(int align) // LEFT, RIGHT, CENTER
FlowLayout(int align, int hgap, int vgap)
```

Example:

```
new FlowLayout(FlowLayout.LEFT, 20, 10);
```

Example:

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

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

        JFrame frame = new JFrame("FlowLayout Example");
        frame.setSize(300, 200);

        // FlowLayout is default, but set for clarity
        frame.setLayout(new FlowLayout());

        frame.add(new JButton("Button 1"));
        frame.add(new JButton("Button 2"));
        frame.add(new JButton("Button 3"));
        frame.add(new JButton("Button 4"));

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

`frame.setLayout(new FlowLayout(FlowLayout.RIGHT));`

CardLayout:

CardLayout is a layout manager that allows you to place multiple components in the same area, but only one component is visible at a time.

It works like a deck of cards — you can switch (flip) between cards.

📦 Package:

`java.awt.CardLayout`

★ When to Use CardLayout

- Login → Dashboard screens
- Wizard forms (Next/Previous)
- Tab-like switching without JTabbedPane
- Multi-page UI in same window

★ Key Methods of CardLayout

Method	Purpose
<code>first(container)</code>	Show first card

last(container)	Show last card
next(container)	Show next card
previous(container)	Show previous card
show(container, "name")	Show card by name

Example (Switch Cards with Buttons)

Example:

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

public class CardLayoutExample extends Applet {
    CardLayout card = new CardLayout();

    public void init() {
        setLayout(card);
        add("One", new Button("Card 1"));
        add("Two", new Button("Card 2"));
        add("Three", new Button("Card 3"));

        // Show the second card
        card.show(this, "Two");
    }
}
```

You control visibility with methods like show(), next(), previous().

Example (Switch Cards with Buttons)

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

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

        JFrame frame = new JFrame("CardLayout Example");
        frame.setSize(400, 250);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create CardLayout and Panel
        CardLayout cardLayout = new CardLayout();
        JPanel cardPanel = new JPanel(cardLayout);

        // Create cards
        JPanel card1 = new JPanel();
        card1.add(new JLabel("This is Card 1"));

        JPanel card2 = new JPanel();
        card2.add(new JLabel("This is Card 2"));
    }
}
```

```

JPanel card3 = new JPanel();
card3.add(new JLabel("This is Card 3"));

// Add cards with names
cardPanel.add(card1, "Card1");
cardPanel.add(card2, "Card2");
cardPanel.add(card3, "Card3");

// Buttons panel
JPanel btnPanel = new JPanel();

JButton nextBtn = new JButton("Next");
JButton prevBtn = new JButton("Previous");

btnPanel.add(prevBtn);
btnPanel.add(nextBtn);

// Button actions
nextBtn.addActionListener(e -> cardLayout.next(cardPanel));
prevBtn.addActionListener(e -> cardLayout.previous(cardPanel));

frame.add(cardPanel, BorderLayout.CENTER);
frame.add(btnPanel, BorderLayout.SOUTH);

frame.setVisible(true);
}
}

```

cardLayout.show(cardPanel, "Card2");

GridLayout

GridLayout is a layout manager that arranges components in a table format (rows × columns).

All components get equal size and are placed in a grid.

 Package:

java.awt.GridLayout

★ Key Features

- Components arranged in rows and columns
- All components have same width & height
- Fills container completely

- No overlap
- Very easy to use

★ Constructor Forms

`GridLayout(rows, cols)`

`GridLayout(rows, cols, hgap, vgap)`

Example:

`new GridLayout(2, 3);`

➡ 2 rows, 3 columns

Example:

```
import javax.swing.*;
import java.awt.*;
public class GridLayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridLayout Example");
        frame.setSize(300, 200);
        // 2 rows, 2 columns
        frame.setLayout(new GridLayout(2, 2));

        frame.add(new JButton("Button 1"));
        frame.add(new JButton("Button 2"));
        frame.add(new JButton("Button 3"));
        frame.add(new JButton("Button 4"));

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

`frame.setLayout(new GridLayout(2, 2, 10, 10));`

Calculator Layout

```
import javax.swing.*;
import java.awt.*;
public class CalculatorGrid {
    public static void main(String[] args) {

        JFrame frame = new JFrame("Calculator");
        frame.setSize(250, 300);

        frame.setLayout(new GridLayout(4, 4, 5, 5));

        for(int i=1; i<=16; i++){
            frame.add(new JButton("" + i));
        }

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

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

★ GridBagLayout in Java

GridBagLayout is the most flexible layout manager in Java.

It arranges components in a grid, but unlike GridLayout, components can have different sizes, different positions, and can span multiple rows/columns.

📦 Package:

java.awt.GridBagLayout

★ Why Use GridBagLayout?

Use it when you want:

- Different component sizes
- Custom alignment
- Components spanning rows/columns
- Professional form design

Example: Login form, registration form, complex UI.

🧠 Basic Idea

GridBagLayout works like a grid with coordinates:

(0,0) (1,0) (2,0)

(0,1) (1,1) (2,1)

★ Important Classes

Class	Use
GridBagLayout	Layout manager
GridBagConstraints	Position rules
Insets	Spacing

★ Important Properties

Property	Meaning
gridx	column position
gridy	row position
gridwidth h	span columns
gridheight ht	span rows
fill	expand size
insets	margin

Example:

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

public class GridBagDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridBagLayout Example");
        frame.setSize(400, 250);

        frame.setLayout(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();

        gbc.insets = new Insets(5,5,5,5);
        // Label
        gbc.gridx = 0;
        gbc.gridy = 0;
        frame.add(new JLabel("Username:"), gbc);
        // TextField
        gbc.gridx = 1;
        frame.add(new JTextField(15), gbc);
        // Label
        gbc.gridx = 0;
        gbc.gridy = 1;
        frame.add(new JLabel("Password:"), gbc);
        // PasswordField
        gbc.gridx = 1;
        frame.add(new JPasswordField(15), gbc);
        // Button
        gbc.gridx = 1;
        gbc.gridy = 2;
        frame.add(new JButton("Login"), gbc);

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

```
}
}
```

★ Example with Component Spanning

```
gbc.gridx = 0;
gbc.gridy = 2;
gbc.gridwidth = 2; // span 2 columns
frame.add(new JButton("Submit"), gbc);
```

★ GridLayout vs GridBagLayout

GridLayout		GridBagLayout	
Equal cells	size	Different allowed	size
Simple		Complex	
Fixed grid		Flexible grid	
Easy coding		More code	

BoxLayout:

BoxLayout in Java is a layout manager that arranges components in a single row or column, either horizontally or vertically. It's part of the javax.swing package and works well when you want a tight, ordered alignment of components.

Example of BoxLayout:

Let's create a vertical layout in an applet:

```
import java.applet.Applet;
import javax.swing.JButton;
import javax.swing.BoxLayout;

public class BoxLayoutExample extends Applet {
    public void init() {
        setLayout(new BoxLayout(this, BoxLayout.X_AXIS)); // Vertical layout

        add(new JButton("Button 1"));
        add(new JButton("Button 2"));
        add(new JButton("Button 3"));
    }
}
```

For horizontal alignment:

```
setLayout(new BorderLayout(this, BorderLayout.X_AXIS)); // Horizontal layout
```

SpringLayout:

SpringLayout in Java is a more advanced and flexible layout manager from the java.awt package. It lets you control the relative positioning of components — like setting fixed or dynamic gaps between them — by using "springs" that can expand or contract.

example:

```
import java.applet.Applet;
import java.awt.Button;
import java.awt.Container;
import javax.swing.SpringLayout;

public class SpringLayoutExample extends Applet {
    public void init() {
        SpringLayout layout = new SpringLayout();
        setLayout(layout);

        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");

        add(btn1);
        add(btn2);

        // Get container for setting layout constraints
        Container container = this;

        // Set constraints for Button 1: 20px from top-left corner
        layout.putConstraint(SpringLayout.WEST, btn1, 20, SpringLayout.WEST, container);
        layout.putConstraint(SpringLayout.NORTH, btn1, 20, SpringLayout.NORTH, container);

        // Set constraints for Button 2: 50px to the right of Button 1 and aligned vertically
        layout.putConstraint(SpringLayout.WEST, btn2, 50, SpringLayout.EAST, btn1);
        layout.putConstraint(SpringLayout.NORTH, btn2, 0, SpringLayout.NORTH, btn1);
    }
}
```

GridLayout:

GridLayout is one of the most powerful and flexible layout managers in Java, mostly used with Swing components. It arranges components both horizontally and vertically, and lets you group them logically — which makes aligning components super easy, especially when building complex forms or responsive layouts.

How GridLayout works:

- Horizontal group → Defines how components align left to right.
- Vertical group → Defines how components align top to bottom.
- Groups can be:
 - Sequential → Components placed one after another.

- Parallel → Components aligned alongside each other.

example:

```
import java.applet.Applet;
import javax.swing.JButton;
import javax.swing.GroupLayout;

public class GroupLayoutExample extends Applet {
    public void init() {
        // Swing components
        JButton btn1 = new JButton("Button 1");
        JButton btn2 = new JButton("Button 2");
        JButton btn3 = new JButton("Button 3");

        // Create GroupLayout and set it for the applet
        GroupLayout layout = new GroupLayout(this);
        setLayout(layout);

        // Automatically set gaps between components
        layout.setAutoCreateGaps(true);
        layout.setAutoCreateContainerGaps(true);

        // Horizontal group: Sequential (left to right)
        layout.setHorizontalGroup(
            layout.createSequentialGroup()
                .addComponent(btn1)
                .addComponent(btn2)
                .addComponent(btn3)
        );

        // Vertical group: Parallel (same row)
        layout.setVerticalGroup(
            layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                .addComponent(btn1)
                .addComponent(btn2)
                .addComponent(btn3)
        );
    }
}
```

NO LAYOUT Manager:

When you use no layout manager in Java (also called "absolute positioning"), you take full manual control of where each component is placed and how big it is. This is done by disabling the layout manager and using the `setBounds(x, y, width, height)` method to specify a component's position and size.

Why use no layout manager?

- Full control over exact positioning of components.
- Useful for fixed-size UIs where resizing is not required.
- Easier for simple, fixed layouts like games or custom designs.

No layout manager (absolute positioning):

```

import java.applet.Applet;
import java.awt.Button;

public class NoLayoutExample extends Applet {
    public void init() {
        // Disable layout manager – we'll use absolute positioning
        setLayout(null);

        // Create buttons
        Button btn1 = new Button("Button 1");
        Button btn2 = new Button("Button 2");
        Button btn3 = new Button("Button 3");

        // Set position and size: (x, y, width, height)
        btn1.setBounds(50, 50, 100, 30);
        btn2.setBounds(200, 50, 100, 30);
        btn3.setBounds(125, 100, 100, 30);

        // Add buttons to the applet
        add(btn1);
        add(btn2);
        add(btn3);
    }
}

```

EVENT HANDLING IN JAVA:

An event is a change in the state of an object triggered by some action such as Clicking a button, Moving the cursor, pressing a key on the keyboard, Scrolling a page, etc. In Java, the `java.awt.event` package provides various event classes to handle these actions.

1. **Foreground Events:** Foreground events are the events that require user interaction to generate. Examples of these events include Button clicks, Scrolling the scrollbar, Moving the cursor, etc.
2. **Background Events:** Events that don't require interactions of users to generate are known as background events. Examples of these events are operating system failures/interrupts, operation completion, etc.

EVENT HANDLING MECHANISM:

Event handling is a mechanism that allows programs to control events and define what should happen when an event occurs. Java uses the **Delegation Event Model** to handle events. This model consists of two main components:

- **Source:** Events are generated from the source. There are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows, etc., to generate events.
- **Listeners:** Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

The two main packages for event handling in Java are:

java.awt.event → Contains classes and interfaces for handling various AWT (Abstract Window Toolkit) events.

javax.swing.event → Contains additional event classes specifically for Swing components.

This package includes event classes and listener interfaces used for handling AWT events. Some commonly used classes and interfaces are:

Event Classes

These classes represent different types of events.

Class	Description
ActionEvent	Generated when a button is clicked or a menu item is selected.
MouseEvent	Generated when a mouse is clicked, pressed, released, moved, or dragged.
ItemEvent	Generated when an item in a list, checkbox, or combo box is selected or deselected.
TextEvent	Generated when the content of a text component (like TextField) changes.
FocusEvent	Generated when a component gains or loses focus.
MouseWheelEvent	It is triggered when the user scrolls the mouse wheel, either up or down.

ActionEvent:

Handling ActionEvent with a JButton in Swing

An ActionEvent occurs when a user interacts with a button, menu item, or other interactive components. In this example, we'll create a simple JButton that changes the text of a label when clicked.

Basic Example: JButton with ActionListener:

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

```

import java.awt.event.ActionListener;
public class ButtonActionExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ActionEvent Example");
        JPanel panel = new JPanel();
        // Create a button and label
        JButton button = new JButton("Click Me");
        JLabel label = new JLabel("Button not clicked");
        // Add ActionListener to the button
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                label.setText("Button Clicked!");
            }
        });
        panel.add(button);
        panel.add(label);
        frame.add(panel);
        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

```

Where ActionEvent is Used:

Component	Action Performed
JButton	Click event
JTextField	Press Enter key
JCheckBox	Select/deselect checkbox
JRadioButton	Select radio button
JComboBox	Select an item from dropdown
JMenuItem	Click menu item

The ItemEvent is triggered when the state of an item in a component changes, such as selecting/deselecting a checkbox or changing a selected item in a combo box.

The ItemListener interface is used to handle ItemEvent.

When to Use ItemEvent?

- You should use ItemEvent when:
- You need to detect checkbox selection/deselection.
- You want to track radio button selection changes.

- You need to respond to combo box item selection.

Example:

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

public class ItemEventAnonymous {
    public static void main(String[] args) {
        // Create frame
        JFrame frame = new JFrame("ItemEvent Example (Anonymous Class)");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);

        // Create checkbox
        JCheckBox checkBox = new JCheckBox("Accept Terms");
        checkBox.setBounds(80, 50, 150, 30);

        // Using Anonymous Class for ItemListener
        checkBox.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {
                if (e.getStateChange() == ItemEvent.SELECTED) {
                    JOptionPane.showMessageDialog(frame, "Checked!");
                } else {
                    JOptionPane.showMessageDialog(frame, "Unchecked!");
                }
            }
        });

        // Add checkbox to frame
        frame.add(checkBox);

        // Set frame visibility
        frame.setVisible(true);
    }
}
```

FocusEvent & FocusListener:

In Java Swing, FocusEvent and FocusListener are used to handle focus-related events in GUI components. These are part of the java.awt.event package.

FocusEvent

FocusEvent is triggered when a component gains or loses focus. It has two main types:

- FocusEvent.FOCUS_GAINED → When the component gets focus.
- FocusEvent.FOCUS_LOST → When the component loses focus.

FocusListener Interface

The FocusListener interface is used to listen for focus events. It has two methods:

- `focusGained(FocusEvent e)`: Called when a component gains focus.
- `focusLost(FocusEvent e)`: Called when a component loses focus.

Example: Handling Focus Events in Swing:

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

public class FocusEventExample extends JFrame {
    public FocusEventExample() {
        setTitle("FocusListener Example");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        JTextField textField1 = new JTextField(15);
        JTextField textField2 = new JTextField(15);
        // Adding FocusListener
        FocusListener focusListener = new FocusListener() {
            @Override
            public void focusGained(FocusEvent e) {
                JTextField source = (JTextField) e.getSource();
                source.setBackground(Color.YELLOW); // Highlight when focused
            }
            @Override
            public void focusLost(FocusEvent e) {
                JTextField source = (JTextField) e.getSource();
                source.setBackground(Color.WHITE); // Reset when focus is lost
            }
        };
        textField1.addFocusListener(focusListener);
        textField2.addFocusListener(focusListener);
        add(new JLabel("Text Field 1:"));
        add(textField1);
        add(new JLabel("Text Field 2:"));
        add(textField2);
        setVisible(true);
    }
    public static void main(String[] args) {
        new FocusEventExample();
    }
}
```

MouseEvent & MouseListener:

In Java Swing, `MouseEvent` and `MouseListener` are used to handle mouse interactions with GUI components. These are part of the `java.awt.event` package.

`MouseEvent`

`MouseEvent` is triggered when the user interacts with the mouse. It includes various actions such as:

- `MOUSE_CLICKED` → When the mouse is clicked (pressed and released).

- MOUSE_PRESSED → When the mouse button is pressed.
- MOUSE_RELEASED → When the mouse button is released.
- MOUSE_ENTERED → When the mouse enters a component's area.
- MOUSE_EXITED → When the mouse exits a component's area.

MouseListener Interface

The MouseListener interface is used to listen for mouse events. It contains the following methods:

- mouseClicked(MouseEvent e): Called when the mouse is clicked.
- mousePressed(MouseEvent e): Called when a mouse button is pressed.
- mouseReleased(MouseEvent e): Called when a mouse button is released.
- mouseEntered(MouseEvent e): Called when the mouse enters a component.
- mouseExited(MouseEvent e): Called when the mouse exits a component.

Example: Handling Mouse Events in Swing:

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

public class MouseEventExample extends JFrame {
    JLabel label;

    public MouseEventExample() {
        setTitle("MouseListener Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        label = new JLabel("Hover or Click Me!");
        label.setOpaque(true);
        label.setBackground(Color.LIGHT_GRAY);
        label.setPreferredSize(new Dimension(200, 50));
        label.setHorizontalAlignment(SwingConstants.CENTER);
        // Adding MouseListener
        label.addMouseListener(new MouseListener() {
            @Override
            public void mouseClicked(MouseEvent e) {
                label.setText("Mouse Clicked!");
                label.setBackground(Color.GREEN);
            }
            @Override
            public void mousePressed(MouseEvent e) {
                label.setText("Mouse Pressed!");
            }
            @Override
            public void mouseReleased(MouseEvent e) {
                label.setText("Mouse Released!");
            }
            @Override
            public void mouseEntered(MouseEvent e) {
                label.setText("Mouse Entered!");
                label.setBackground(Color.YELLOW);
            }
            @Override
            public void mouseExited(MouseEvent e) {
                label.setText("Mouse Exited!");
                label.setBackground(Color.LIGHT_GRAY);
            }
        });
    }
}
```

```

    });

    add(label);
    setVisible(true);
}

public static void main(String[] args) {
    new MouseEventExample();
}
}

```

MouseEvent:

MouseEvent is triggered when the user rotates the mouse wheel. It is part of the java.awt.event package and is used with MouseWheelListener.

MouseEvent Class

The MouseEvent provides information about the scroll event, such as:

- getWheelRotation() → Returns -1 for upward scrolling and 1 for downward scrolling.
- getScrollType() → Determines whether the scrolling is based on units or blocks.
- getScrollAmount() → Returns the number of units to scroll.

Example: Handling Mouse Wheel Events:

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseWheelEvent;
import java.awt.event.MouseWheelListener;

public class MouseWheelExample extends JFrame {
    JLabel label;
    int fontSize = 16; // Initial font size

    public MouseWheelExample() {
        setTitle("MouseWheelListener Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        label = new JLabel("Scroll to Resize Text!");
        label.setFont(new Font("Arial", Font.PLAIN, fontSize));

        // Adding MouseWheelListener
        addMouseWheelListener(new MouseWheelListener() {
            @Override
            public void mouseWheelMoved(MouseWheelEvent e) {
                int rotation = e.getWheelRotation(); // -1 for up, 1 for down
                if (rotation < 0) {
                    fontSize += 2; // Increase font size
                } else {
                    fontSize -= 2; // Decrease font size
                }
                fontSize = Math.max(10, fontSize); // Prevent too small text
                label.setFont(new Font("Arial", Font.PLAIN, fontSize));
            }
        });

        add(label);
    }
}

```

```

        setVisible(true);
    }

    public static void main(String[] args) {
        new MouseWheelExample();
    }
}

```

TextEvent & TextListener:

In Java, TextEvent and TextListener are used to monitor changes in text components like TextField and TextArea. However, these are part of AWT (java.awt.event package) and do not work directly with Swing components like JTextField or JTextArea.

In Swing, we use DocumentListener instead of TextListener.

TextEvent Class (AWT)

TextEvent is triggered when the text changes in an AWT component (TextField or TextArea). It does not differentiate between typing, pasting, or deletion.

Example: Handling Text Changes

```

import javax.swing.*;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;
import java.awt.*;

public class TextEventExampleJFrame extends JFrame {
    public TextEventExampleJFrame() {
        setTitle("TextEvent with JFrame");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        JTextField textField = new JTextField(20);
        JLabel label = new JLabel("Type something...");

        // Adding DocumentListener to JTextField
        textField.getDocument().addDocumentListener(new DocumentListener() {
            @Override
            public void insertUpdate(DocumentEvent e) {
                label.setText("You typed: " + textField.getText());
            }

            @Override
            public void removeUpdate(DocumentEvent e) {
                label.setText("You typed: " + textField.getText());
            }

            @Override
            public void changedUpdate(DocumentEvent e) {
                // Not needed for plain JTextField
            }
        });

        add(textField);
        add(label);

        setVisible(true);
    }

    public static void main(String[] args) {

```

```

        new TextEventExampleJFrame();
    }
}

```

WindowListener:

WindowListener is used to handle window-related events in a Swing application. It is part of the java.awt.event package and is commonly used with JFrame to manage window actions like opening, closing, minimizing, etc.

WindowListener Interface

The WindowListener interface contains the following methods:

Method	Description
windowOpened(WindowEvent e)	Called when the window is first opened.
windowClosing(WindowEvent e)	Called when the user tries to close the window.
windowClosed(WindowEvent e)	Called after the window has been closed.
windowIconified(WindowEvent e)	Called when the window is minimized.
windowDeiconified(WindowEvent e)	Called when the window is restored from minimized.
windowActivated(WindowEvent e)	Called when the window becomes active.
windowDeactivated(WindowEvent e)	Called when the window loses focus.

Handling Window Events

```

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

public class WindowListenerExample extends JFrame {
    public WindowListenerExample() {
        setTitle("WindowListener Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE); // Prevents direct closing

        JLabel label = new JLabel("Try minimizing, closing, or activating the window.",
        SwingConstants.CENTER);
        add(label);

        // Adding WindowListener to JFrame
        addWindowListener(new WindowListener() {
            @Override
            public void windowOpened(WindowEvent e) {
                System.out.println("Window Opened");
            }

            @Override
            public void windowClosing(WindowEvent e) {
                int choice = JOptionPane.showConfirmDialog(null, "Are you sure you want to
                close?", "Confirm Exit",
                JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE);
                if (choice == JOptionPane.YES_OPTION) {
                    dispose(); // Closes the window
                }
            }
        });
    }
}

```

```

    }
}

@Override
public void windowClosed(WindowEvent e) {
    System.out.println("Window Closed");
}

@Override
public void windowIconified(WindowEvent e) {
    System.out.println("Window Minimized");
}

@Override
public void windowDeiconified(WindowEvent e) {
    System.out.println("Window Restored");
}

@Override
public void windowActivated(WindowEvent e) {
    System.out.println("Window Activated");
}

@Override
public void windowDeactivated(WindowEvent e) {
    System.out.println("Window Deactivated");
}
});

setVisible(true);
}

public static void main(String[] args) {
    new WindowListenerExample();
}
}

```

Adaptor Classes in Java

Adapter classes in Java provide default implementations for interface methods. They are useful when you need to implement an interface but don't want to override all its methods.

Why Use Adapter Classes?

Some interfaces in Java contain multiple methods, and if a class implements such an interface, it must provide implementations for all the methods. However, in many cases, we may not need all of

them. Adapter classes help by providing default (empty) implementations, allowing us to override only the required methods.

Common Adapter Classes in Java

Java provides several built-in adapter classes, mainly in the `java.awt.event` package. Some of the commonly used adapter classes include:

Adapter Class	Implements Interface	Use Case
KeyAdapter	KeyListener	Handle keyboard events
MouseAdapter	MouseListener	Handle mouse click, press, and release events
MouseMotionAdapter	MouseMotionListener	Handle mouse movement events
WindowAdapter	WindowListener	Handle window events like opening, closing
FocusAdapter	FocusListener	Handle focus gained/lost events
ComponentAdapter	ComponentListener	Handle component resize, move events

Example of KeyAdapter class:

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

public class KeyAdapterSwingExample {
    public static void main(String[] args) {
        // Create JFrame
        JFrame frame = new JFrame("KeyAdapter Example");
        frame.setSize(400, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
        // Create JTextField
        JTextField textField = new JTextField();
        textField.setBounds(50, 50, 200, 30);
        // Add KeyAdapter to handle key events
        textField.addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                System.out.println("Key Pressed: " + e.getKeyChar());
            }
        });
        // Add components to frame
        frame.add(textField);
        frame.setVisible(true);
    }
}
```

KeyListener Example in Swing (Without Adapter Class)

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

public class KeyListenerExample implements KeyListener {
    JTextField textField;

    KeyListenerExample() {
        // Create JFrame
        JFrame frame = new JFrame("KeyListener Example");
```

```

frame.setSize(400, 200);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setLayout(null);

// Create JTextField
textField = new JTextField();
textField.setBounds(50, 50, 200, 30);

// Add KeyListener directly
textField.addKeyListener(this);

// Add components to frame
frame.add(textField);
frame.setVisible(true);
}

// Implement KeyListener methods
public void keyPressed(KeyEvent e) {
    System.out.println("Key Pressed: " + e.getKeyChar());
}

public void keyReleased(KeyEvent e) {
    // Not used, but must be implemented
}

public void keyTyped(KeyEvent e) {
    // Not used, but must be implemented
}

public static void main(String[] args) {
    new KeyListenerExample();
}
}

```

MouseListener Example in Swing

```

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

public class MouseAdapterExample {
    public static void main(String[] args) {
        // Create JFrame
        JFrame frame = new JFrame("MouseListener Example");
    }
}

```

```

frame.setSize(400, 300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setLayout(null);

// Create JLabel
JLabel label = new JLabel("Click anywhere!", SwingConstants.CENTER);
label.setBounds(100, 100, 200, 50);
frame.add(label);

// Add MouseAdapter
frame.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        label.setText("Mouse Clicked at X: " + e.getX() + ", Y: " + e.getY());
    }
});

// Show Frame
frame.setVisible(true);
}

```

*****All the Best*****

Unit 1 Practical Questions:

1. Write a Java program to declare and initialize variables of Integer, Float, Character, and Boolean types. Demonstrate the use of Keywords, Literals, Identifiers, Whitespace, Separators, and Comments in the program.
2. Create a Java program that takes two numbers as input and performs addition, subtraction, multiplication, division, modulus, relational comparisons, and logical operations.

3. Create a Java program that takes two numbers as input and performs addition, subtraction, multiplication, division, modulus, relational comparisons, and logical operations.
4. Write a Java program to demonstrate local, instance, and static variables in a class.
5. Create a program that takes user input for age and prints:

Unit 2 Practical Questions

1. Write a Java program to demonstrate the usage of Math class methods (e.g., sqrt(), pow(), abs()) and String class methods (length(), charAt(), substring()).
2. Implement a program that takes user input using Scanner, generates random numbers using Random, stores elements in Vector, and demonstrates stack operations using Stack.
3. Implement a program that throws and catches a user-defined exception when the user enters a negative number.
4. Write a Java program to demonstrate public, private, protected, and default access specifiers in an inheritance hierarchy.
5. Create a Vehicle class with a speed() method. Inherit it in Car and Bike classes, overriding the speed() method.
6. Write a Java program with a nested class inside a main class and access its methods.
7. Create an abstract class Shape with an abstract method draw(). Implement it in subclasses Circle and Rectangle. Also, demonstrate the use of a final class.

Unit 3 Practical questions

1. Creating a Simple Thread (Thread Life Cycle) : Understand thread creation and the different states of a thread.
2. Using Thread Class and Its Methods : Demonstrate the use of setName(), getName(), setPriority(), and getPriority().
3. Implementing Multithreading : Prevent race conditions.
4. Demonstrating Daemon and Non-Daemon Threads : Show the difference between daemon and non-daemon threads.
5. Implementing Runnable Interface for Multithreading : Implement multithreading using Runnable instead of Thread.
6. Read Data from a File (Using FileInputStream - Byte Stream): Demonstrate how to read bytes from a file using FileInputStream.
7. Writing Data to a File Using FileOutputStream: Demonstrate writing a file using FileOutputStream (Byte Stream).
8. Reading a File Using FileReader (Character Stream) : Demonstrate reading a file using FileReader (Character Stream).
9. Writing Data Using FileWriter (Character Stream) : Demonstrate writing a file using FileWriter (Character Stream).
10. Understanding the Applet Life Cycle : Demonstrate the different stages of the Applet Life Cycle (init(), start(), paint(), stop(), destroy()).
11. Implementing and Executing an Applet with Parameters : Pass parameters from an HTML file to an Applet.
12. Playing an Audio File in an Applet : Demonstrate how to play an audio file using Applet's AudioClip.
13. Drawing Shapes Using the Graphics Class : Use the Graphics class to draw basic shapes.
14. Using FlowLayout : Demonstrate FlowLayout, where components are arranged in a row and wrap to the next line if necessary.
15. Using BorderLayout : Demonstrate BorderLayout, which places components in NORTH, SOUTH, EAST, WEST, and CENTER regions.

16. Using CardLayout : Demonstrate CardLayout, where multiple panels are stacked and switched dynamically.
17. Using GridLayout : Demonstrate GridLayout, where components are arranged in a grid of rows and columns.
18. Using GridBagLayout : Demonstrate GridBagLayout, which allows flexible grid-based component positioning.

Unit 4 Practical questions:

1. Create a Student Registration Form using Swing components like JTextField, JPasswordField, JRadioButton, JCheckBox, and JButton. Implement event handling for the Submit button.
2. Create a simple calculator using JTextField for input and JButton for addition, subtraction, multiplication, and division. Use ActionListener to handle button clicks.
3. Create a GUI application with a menu bar containing options like File, Edit, and Help using JMenuBar, JMenu, and JMenuItem.
4. Create an application that tracks mouse events (click, press, release, move, and drag) using MouseListener and MouseMotionListener.
5. Demonstrate the use of JProgressBar by creating a progress loading simulation.

BKNMU Old Papers

1. [April/May -2022 \[OLD COURSE\] CS-19 PROGRAMMING WITH JAVA \(CORE\).](#)
2. [March/April-2023 \[NEW COURSE\] Programming with JAVA\(Core\).](#)
3. [March/April-2023 \[OLD COURSE\] CS-19 PROGRAMMING WITH JAVA\(CORE\).](#)
4. [March/April-2024 \[NEW COURSE\] Programming with JAVA \(Core\).](#)
5. [March/April-2024 \[OLD COURSE\] CS-19 PROGRAMMING WITH JAVA\(CORE\).](#)