<Java Programming>

)

A project Report submitted to

DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY

BACHELOR OF COMPUTER SCIENCE (BSC)

By
ALKA MARKAM

Roll No.: 22070108
Enrollment No.:
GGV/22/05108

Under the Guidance of
Dr. Ratnesh Srivastava

DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY

GURU GHASIDAS VISHWAVIDYALAYA, BILASPUR

Session: 2022-2025

CERTIFICATE OF SUPERVISOR(S) /GUIDE

This is to certify that the work incorporated in the project **Java Programming** is a record of six month project work assigned by our Industry/Company/Institution, successfully carried out by Alka Markam bearing Enrollment No. GGV/22/05108 under my guidance and supervision for the award of Degree of Bachelor of Computer Science (BSC) of **DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY, GURU GHASIDAS VISHWAVIDYALAYA, BILASPURC.G., INDIA**. To the best of my knowledge and belief the report embodies the work of the candidate him/herself and has duly been successfully completed.

Signature of the Supervisor/Guide

Name: Dr. Ratnesh Shrivastava

Signature of HOD

Name: Dr. Ratnesh Shrivastava

DELCLARATION BY THE CANDIDATE

I, **Alka Markam**, Student of VI Semester BSC,**DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY, GURU GHASIDAS VISHWAVIDYALAYA, BILASPUR**, bearing Enrolment Number **GGV/22/05108** here by declare that the project titled **Java Programming** has been carried out by me under the Guidance/Supervision of **Dr. Ratnesh Shrivastava**, submitted in partial fulfillment of the requirements for the award of the Degree of Bachelor of Computer Science (BSC) by the Department Of Computer Science & Information Technology, Guru Ghasidas Vishwavidyalaya, Bilaspur during the academic year 2022-25 .This report has not been submitted to any other
Organization/University for any award of Degree/Diploma.

(Signature of Candidate)

Date: 08-05-2025

Place: Bilaspur

ACKNOWLEDGEMENT

I have great pleasure in the submission of this project report entitled **Java Programming** in partial fulfillment of the degree of Master of Computer Applications. While Submitting this Project report, I take this opportunity to thank those directly or indirectly related to project work.

I would like to thank my guide **Dr. Ratnesh Shrivastava, who** has provided the opportunity and organizing project for me. Without his active co-operation and guidance, it would have become very difficult to complete task in time.

I would like to express sincere thanks to Dr. Ratnesh Shrivastava, Head of Department of Computer Science & Information Technology, Guru Ghasidas Vishwavidyalaya, Bilaspur C.G

Acknowledgement is due to my parents, family members, friends and all those persons who have helped me directly or indirectly in the successful completion of the project work.

NAME OF STUDENT:
Alka Markam

Table of Content

Abstract

This project presents a concise yet comprehensive exploration of essential and intermediate Java programming concepts aimed at enhancing both theoretical understanding and practical application. Java is one of the most widely used
programming languages in the world. Java, being a platform-independent, object-oriented language, remains a dominant force in software development due to its robust architecture, security features, and vast ecosystem. The content of this project is structured around key programming domains that form the backbone of Java development. It begins with **exception handling**, a mechanism that ensures
reliability and stability by allowing graceful error management. The next segment explores **multithreading**, enabling programs to perform multiple operations
concurrently, which is crucial for developing responsive and high-performance applications. The section on **socket programming** introduces network-based communication using Java's Socket and Server Socket classes, helping learners understand how real-time data transmission works in distributed systems. This is followed by an overview of **Servlets and JSP**, which are vital components for building dynamic and interactive web applications in Java, demonstrating how Java powers the server-side of web technologies. Lastly, the project delves into **Swing**, a GUI toolkit that facilitates the creation of rich desktop interfaces through interactive components and event-driven programming. By combining these topics into a single, coherent note, this project serves as a practical guide for learners and aspiring developers to grasp the foundational and applied aspects of Java development, setting the stage for more advanced studies or real-world projects.

Introduction

1. Introduction

Java has become a cornerstone of modern software development due to its versatility, scalability, and robustness. Whether developing standalone desktop applications or enterprise-level web systems, Java offers developers the tools needed for efficient and secure programming. This paper serves as a guide for understanding essential and intermediate Java concepts, aimed at learners who wish to build a solid foundation and enhance their practical skills.

---

2. Exception Handling in Java

Exception handling is a mechanism used to handle runtime errors, allowing a program to continue execution rather than crashing. Java provides a robust framework for managing exceptions using try, catch, finally, and throw/throws keywords. There are two types of exceptions: checked and unchecked. Handling these effectively ensures the program behaves predictably under abnormal conditions. For example, a program that attempts to divide a number by zero can catch the ArithmeticException and display a user-friendly message instead of terminating abruptly.

---

3. Multithreading in Java

Multithreading is the capability of a CPU or a single core in a multi-core processor to execute multiple threads concurrently. In Java, threads can be created by extending the Thread class or implementing the Runnable interface. This allows programs to perform multiple operations simultaneously, improving efficiency, especially in real-time applications such as games or simulations. Key concepts like thread lifecycle, synchronization, and inter-thread communication play a crucial role in managing concurrent execution safely and effectively.

---

4. Socket Programming in Java

Socket programming enables communication between computers over a network. Java provides the Socket and ServerSocket classes to facilitate client-server communication. A typical socket program involves a server that listens for client

requests and a client that initiates a connection. This forms the basis for many networked applications, including chat systems and file sharing tools. Implementing a basic socket program introduces learners to core networking concepts such as ports, IP addresses, and data streams.

---

5. Servlet and JSP using Java

Java Servlets and JSP (JavaServer Pages) are crucial for developing dynamic web applications. A Servlet is a Java class that handles HTTP requests and responses on the server side. It follows a specific lifecycle and is managed by a servlet container like Apache Tomcat. JSP, on the other hand, allows embedding Java code directly into HTML, making web development more intuitive. Together, Servlets and JSP provide a powerful model for building responsive, data-driven web interfaces. A typical use case involves a form submission handled by a Servlet, which then dynamically generates a response using JSP.

---

6. Using Swing in Java

Swing is a part of Java's standard library used for creating rich graphical user interfaces. Unlike command-line programs, Swing applications offer buttons, text fields, menus, and other interactive elements. Components such as JFrame, JPanel, and JButton help structure the interface, while event listeners respond to user actions. Swing applications are useful for building calculators, form-based software, and even small-scale games. A hands-on Swing project not only improves programming skills but also introduces the principles of event-driven programming

Literature Review

Java, developed by Sun Microsystems in 1995 and now maintained by Oracle Corporation, has consistently remained one of the most influential programming languages in both academic and industrial domains. Its object-oriented paradigm, platform independence via the Java Virtual Machine (JVM), and extensive standard libraries have contributed to its widespread adoption. The literature surrounding Java programming underscores its significance in application development, web technologies, enterprise solutions, and mobile platforms.

Java's "write once, run anywhere" philosophy is made possible by the JVM, which abstracts the underlying hardware and operating system. This portability has made Java an attractive option for cross-platform application development. As noted by Horstmann and Cornell (2008), Java's strict compile-time and runtime checking, along with automatic memory management through garbage collection, reduces common programming errors and improves program reliability.

In object-oriented programming (OOP), Java emphasizes encapsulation, inheritance, and polymorphism, which are fundamental principles for code reuse, modularity, and scalability. According to Liang (2015), Java's class-based structure and access control mechanisms promote strong software architecture, especially for large-scale applications.

Exception handling is another feature that enhances the robustness of Java applications. By allowing programmers to catch and manage runtime errors without disrupting program flow, Java improves user experience and system stability. Research by Eckel (2006) highlights how Java's structured exception mechanism (try, catch, finally) helps isolate error-handling logic from business logic, making code more organized and readable.

Java's multithreading capability is one of its strongest features, especially relevant in the age of multi-core processors. The ability to execute multiple threads simultaneously allows developers to create responsive user interfaces and efficient server-side applications. Goetz et al. (2006) emphasize the importance of proper synchronization and thread-safe data structures to avoid concurrency issues, highlighting Java's advanced features such as the java.util.concurrent package.

On the networking front, Java supports socket programming, RMI (Remote Method Invocation), and HTTP-based communication, making it well-suited for distributed systems. Java's java.net package provides a high-level abstraction for both TCP and UDP communications. Literature shows that Java's networking APIs are extensively used in the development of chat systems, multiplayer games, IoT solutions, and real-time applications.

In the domain of GUI development, Java's Swing and JavaFX libraries provide rich user interface capabilities. Swing allows the creation of cross-platform desktop applications with sophisticated components. Though now somewhat overshadowed by JavaFX, both remain important in educational and legacy systems.

Problem Statement

In the current digital age, software systems demand high performance, reliability, and interactivity across multiple platforms. While Java remains one of the most widely used programming languages, learners often struggle to bridge the gap between theoretical knowledge and practical implementation. Despite the availability of resources, there is a lack of comprehensive, integrative programming examples that combine core Java concepts such as exception handling, multithreading, socket programming, Servlets, JSP, and Swing into a unified system.

This research addresses the challenge of creating a compressive Java application that demonstrates these diverse features working together in a real-world context. The goal is to identify common obstacles faced by developers during implementation—such as concurrency issues, network communication complexities, UI responsiveness, and server-side logic—and to provide effective solutions through a modular and integrated Java program.

Motivation

In today's rapidly evolving technological landscape, programming languages play a crucial role in software development, system automation, and enterprise solutions.
Among these, Java stands out due to its platform independence, object-oriented features, and wide applicability across web, desktop, and network-based applications. The motivation behind choosing integrated Java programming as the focus of this research lies in its versatility, robustness, and continued relevance in both academic learning and industry applications.

As a Master of Computer Applications (MCA) student, gaining hands-on experience with core Java concepts such as exception handling, multithreading, socket programming, Servlets, JSP, and Swing is essential. These areas not only demonstrate Java's comprehensive capabilities but also represent real-world challenges and scenarios faced by developers. By building and integrating these components, this project aims to simulate the kind of multi-faceted systems found in enterprise-level software.

Furthermore, this research project serves as a platform to bridge theoretical knowledge with practical skills. Developing a compressive Java program that brings together different features provides a deeper understanding of how modular code, user interaction, concurrency, and network communication can coexist within a unified application. It also fosters logical thinking, design planning, and debugging expertise—key attributes for any software developer.

Lastly, the motivation is driven by the desire to create a meaningful, well-rounded project that not only fulfills academic requirements but also enhances employability and readiness for the software industry. Through this project, the goal is to demonstrate technical proficiency in Java and contribute to a portfolio of real-world programming solutions.

Objectives

The primary objective of this report is to develop a comprehensive understanding of core and advanced Java programming concepts through practical implementation.
The paper aims to explore essential Java programming paradigms including exception handling, multithreading, socket programming, servlet and JSP-based web development, and graphical user interface creation using Swing. By implementing these components in real-time programs, the report seeks to highlight how Java can be utilized to build robust, efficient, and scalable software solutions.

The specific goals of the report are as follows:

1. To introduce and explain core Java concepts such as variables, data types, control structures, arrays, and object-oriented principles like encapsulation, inheritance, polymorphism, and abstraction.

2. To offer a detailed understanding of advanced Java topics, including exception handling, file handling, multithreading, synchronization, GUI development using AWT/ Swing, and applet programming.

3. To support incremental learning by arranging topics in a logical sequence, enabling learners to gradually progress from basic constructs to more complex programming techniques.

4. To include hands-on code examples and explanations that help learners visualize and apply the theoretical concepts in real-world programming scenarios.

5. To enhance self-learning and revision capabilities by presenting concise notes that can be used for quick reference during exams, interviews, or coding tasks.

6. To serve as a teaching aid for instructors and educators who wish to use a unified and reliable material for classroom instruction or lab sessions.

7. To foster confidence and practical skill development among learners, enabling them to write efficient Java programs and tackle programming challenges effectively.

In essence, the report is designed to be more than just a study material—it is a comprehensive learning companion that supports continuous growth and practical application in the field of Java programming

Methodology

1. Topic Selection and Research:
Identify and study key Java programming areas such as exception handling, multithreading, networking (sockets), web development (Servlets and JSP), and GUI development (Swing).

2. Environment Setup:
Configure the development environment using JDK (Java Development Kit) and an IDE such as Eclipse or IntelliJ IDEA.

3. Program Development:

Implement standalone Java programs to demonstrate exception handling and multithreading concepts.
Develop client-server applications to understand socket programming.
Design and deploy basic web applications using Servlets and JSP. Create desktop applications using Swing for GUI-based interactions.

4. Code Integration: For each topic, relevant Java code snippets were created and included to demonstrate practical implementation. These examples were tested for correctness and clarity, ensuring they are suitable for beginners.

5. Simplification and Clarification: Technical content was rewritten in simple language where needed, to make it accessible for students with varying levels of programming background.

6. Review and Refinement: The content was reviewed to ensure consistency, accuracy, and completeness. Redundant or overly complex explanations were simplified to maintain focus and flow.

Result

This project successfully enhanced understanding of key Java concepts through practical implementation. Topics like exception handling, multithreading, socket programming, Servlets, JSP, and Swing were explored with real examples. As a result, foundational and applied programming skills were developed, providing confidence to build functional applications and paving the way for advanced learning in Java-based technologies. The hands-on approach helped in strengthening logic building, understanding program flow, and improving debugging techniques. Additionally, the integration of both core and web-based Java concepts allowed for a well-rounded experience, preparing the learner to confidently approach both
academic projects and industry-level software development tasks.

Conclusion

This project offers a concise yet meaningful journey through core and advanced Java programming concepts, highlighting the practical importance of each topic. From handling exceptions to building multithreaded applications, and from enabling network communication through socket programming to designing dynamic web pages using Servlets and JSP, each section contributes to a well-rounded
understanding of Java. The inclusion of Swing for GUI development adds a visual and interactive aspect, showcasing Java's versatility across different application types.

Together, these topics provide a solid foundation for building real-world software solutions and prepare learners to explore more advanced areas such as Spring Framework, Hibernate, or mobile app development with Android. By combining theoretical insights with practical examples, this project not only enhances technical skills but also encourages a deeper appreciation for Java's role in modern software engineering.

References

1.Effective Java by Joshua Bloch

(A highly recommended book for writing robust, maintainable, and efficient Java code.)

2.Java: The Complete Reference by Herbert Schildt

(A comprehensive book covering basic to advanced Java concepts with examples.)

3.GeeksforGeeks – Java Programming Language

https://www.geeksforgeeks.org/

(Offers tutorials, problem-solving, and practical examples for learners at all levels.)

4.YouTube Channels (for visual learners)

- LearnCoding
- Telusko

5.TutorialsPoint Java Programming

https://www.tutorialspoint.com/java/

6. Java Socket Programming Examples –

https://www.baeldung.com/a-guide-to-java-sockets

JAVA

Java is a class based high-level, object-oriented programming language developed by “James Gosling” and his team in the year 1995.
Java is known for its platform independence , which means that java programs can run on any device that has a java virtual machine (JVM) installed. This makes Java a popular choice for developing cross-platform applications.

NOTE:-
1.The first version of the java(JDK 1.0) was released on the year Jan- 23rd 1996 by “Sun MIcrosoft”.
2.Latest version of java (JDK 16) on the day 16th march 2021 by “Orack”

Syntax:- class

```
class_name
{
   public static void main(String args[])
   {
       //code
   }
}
```

How java works:-
Java is compiled into the bytecode and then it is interpreted to machine code

JDK(Java Development Kit)- collection of tools used for developing and running java programming.

JRE(Java Runtime Environment)- helps in executing programs developed in java.

JVM:- JVMacts as a run time engine to run java application jvm is actuallyy the one that actually calls the main method. JVM is a part of JRE(Java Runtime Environment).
Java application called WORA(Write Once Run Anywhere) this means a programmer can devel java code on system and expect it to any other java adjustment. This all possible because of JVM.

OOPs in java
Oops stands for object oriented programming language, the main purpose of oop is to deal with real world entities using programming language.

Features:-

1. **Class:-** class is a collection of object and it doesn’ttake any space in memory, class is also called a blueprint/logical entity.

Class

Pre-defined	**User-defined**
1. Scanner	1. Dog
2. Console	2.A
3. System	3.Test
4. String	

2. Object:- An object consist of a class that has actual values for the properties defined in the class. An objectvrepresents a specific entity with a defined state and behavior.
When you create an object, you allocate memory for it and initialize its attributes.

Syntax:- class_name obj_name = new class_name()

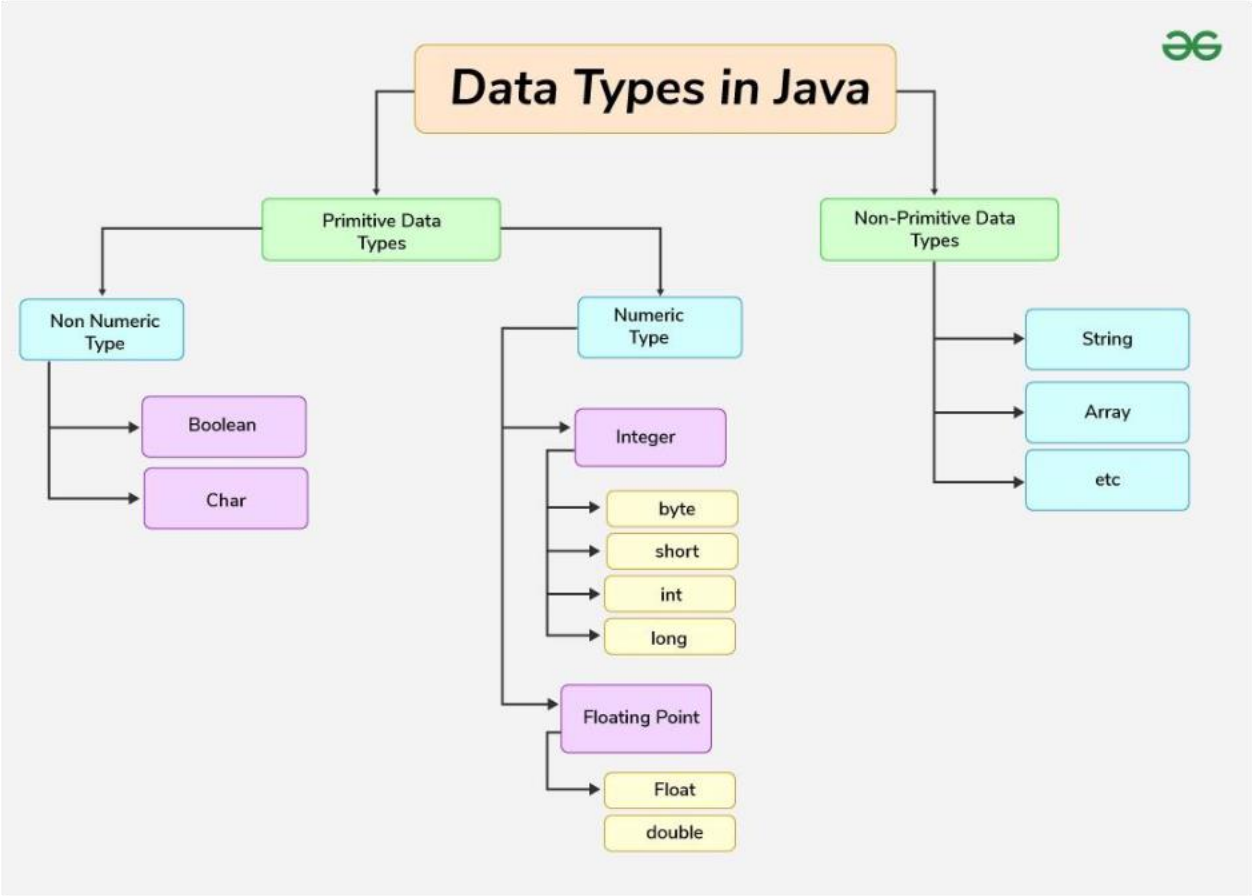
Variables:- a variable is a container used to hold data. The value can be changed by the execution of the program.

Ex:- int number = 8; value it stores
Data types variable name

Rules for declaring a variable name:-
We can choose a name which declaring a java variable if the following rules are followed 1. Must not begin with a digit int 1array; invalid

2. Name is case sensitive.
3. Should not be keyword.
4. White space not allowed.
5. Can contain alphabets, characters, -characters, and digits if the other conditions are met.

DATA TYPES- specified the different type of values that are stored on the variables.



Primitive Data types:- Primitive data types are the types in java that can store a single value and do not provide any special capability.

- **Byte** -value range from -128 to 127
  -Take 1 byte(8 bytes) -default value is 0
- **Short-** value ranges from -32768 to 32767
  -Takes 2 bytes
  -default value is 0
- **Int**-value ranges from (-2^31) to (2^31 -1).
  -Takes 4 bytes
  -default value is 0
- **float**-stores fractional numbers ranging from 3.4e-038 to 3.4e+038 -takes 8 bytes -default value is 0.0f
- **long** -value ranges from (-2^63) to (2^63 -1).
  -Takes 8 bytes
  -default value is 0

- **double** -value ranges from(1.7e-308 to 1.7e+308)
  -Takes 8 bytes
  -default value is 0.0d
- **char** -value ranges from 0 to 6.5535()
  -Takes 2 bytes
  -default value is 0
- **Boolean** -value can be T or F
  -size depends on JVM
  -default value is false

Non-Primitive Data Types:- These are the variable size and are usually declared with a 'new' keyword.
Eg: string, Arrays

Literals:- a constant value which can be assigned to the variable is called as a literal.
101- int literal
10.1f- float literal
10.1- double literal
'A'-char literal
True- boolean literal
"Harry"- string literal

Keywords:- wordswhich are reserved and used by the java compiler. They can not be used as identifier.

Reading data from keyword
In order to read data from the keyboard, Java has a scanner class. Scanner class has a lot of methods to read the data from the keyboard.

Scanner s = newscanner(system.in); read from the keyboard Int a =
s.nextint() method to read from the keyboard

User input in java

```
package com.company
import java.util.Scanner;

public class CWH_OS_Taking Input{
        public static void main (string[] args) {
        System out println ("Taking Input from the User");
```

```
            Scanner sc = newscanner(System.in);
System.out.println("Enter number 1");            int a =
sc.nextInt();
            system. out println ("enter number 2”);
int b = sc.nextInt();            sum = a+b;
           system.out.println("The sum of these number is”);
system.out.println(sum);
            }
}
```

Output
Taking input from the
Enter number 1 40
Enter number 2
50
The sum of these number
90

OPERATORS IN JAVA:-
Here are the different types of operators in Java:

Arithmetic Operators:- are used to perform simple arithmetic operations on primitive and nonprimitive data types.
1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
5. Modulus (%)
6. Increment (++): increases the value of a variable by 1 7.
Decrement (–): decreases the value of a variable by 1

Assignment Operators:-
1. Assignment (=): assigns a value to a variable
2. Addition Assignment (+=): adds a value to a variable and assigns the result
3. Subtraction Assignment (-=): subtracts a value from a variable and assigns the result
4. Multiplication Assignment (*=): multiplies a value with a variable and assigns the result
5. Division Assignment (/=): divides a variable by a value and assigns the result
6. Modulus Assignment (%=): calculates the modulus of a variable and a value and assigns the result

Comparison Operators:-

1. Equal To (==): checks if two values are equal
2. Not Equal To (!=): checks if two values are not equal
3. Greater Than (>): checks if a value is greater than another
4. Less Than (<): checks if a value is less than another
5. Greater Than or Equal To (>=): checks if a value is greater than or equal to another 6. Less Than or Equal To (<=): checks if a value is less than or equal to another

Logical Operators:-

1. Logical AND (&&): checks if both conditions are true 2.
Logical OR (||): checks if at least one condition is true 3. Logical
NOT (!): reverses the result of a condition

Bitwise Operators:-

1. Bitwise AND (&): performs a binary AND operation on two integers
2.Bitwise OR (|): performs a binary OR operation on two integers
3. Bitwise XOR (^): performs a binary XOR operation on two integers
4. Bitwise Complement (~): flips the bits of an integer
5. Left Shift (<<): shifts the bits of an integer to the left
6. Right Shift (>>): shifts the bits of an integer to the right
7. Unsigned Right Shift (>>>): shifts the bits of an integer to the right without sign extension

Ternary Operator:- 1. Ternary Operator (?:): evaluates a condition and returns one of two values based on result.

EXCEPTION HANDLING IN JAVA

An exception is unexpected/unwanted/abnormal situation that occurred at runtime called exception. Ex:- 1) powercut exception
2)FileNotFoundException

WHAT IS EXCEPTION HANDLING?

In exception handling, we should have an alternate source through which we can handle the exception.
Exception in Java is an error condition that occurs when something wrong happens during the program execution.

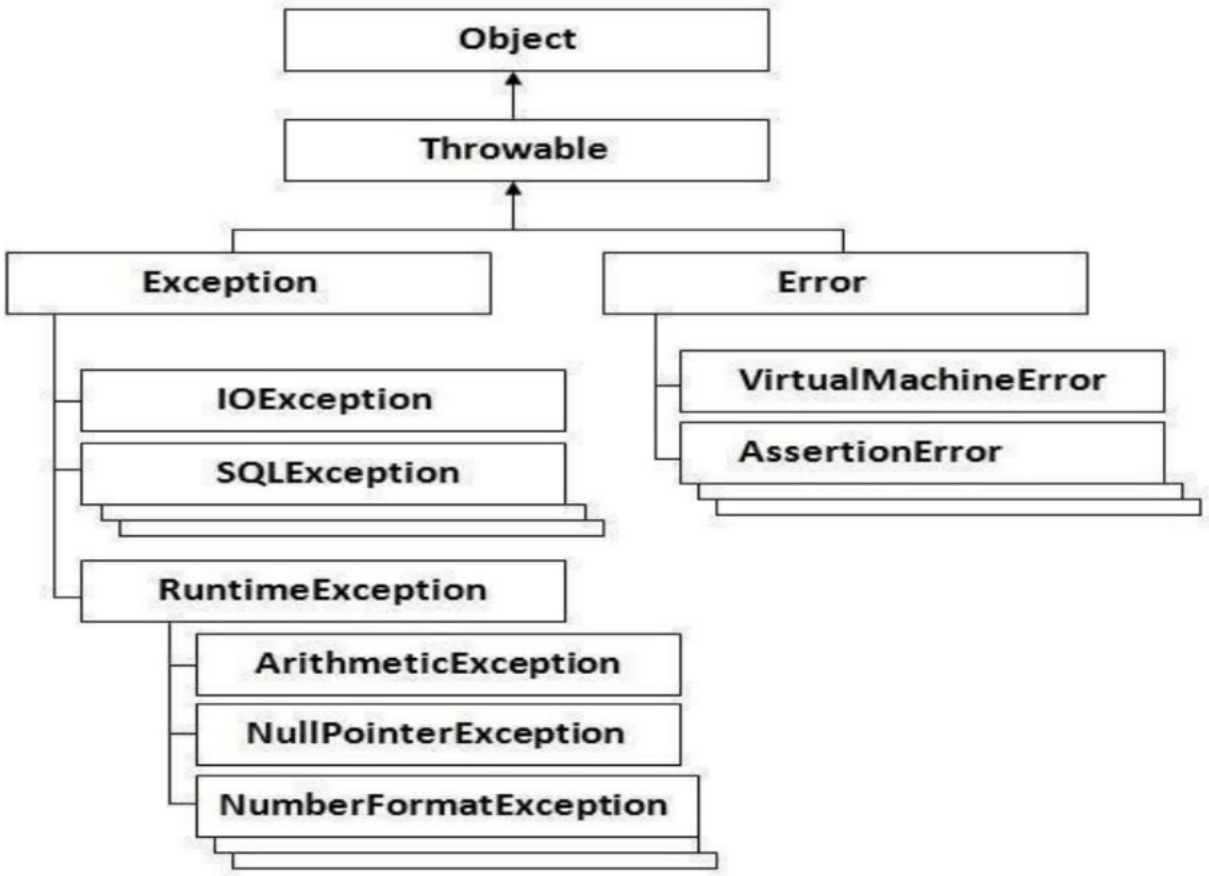
Example: Showing an Arithmetic Exception or you can say divide by zero exception. import java.io.*;

```
class Geeks {  public static void
main(String[] args)
 {
 int n = 10;  int m = 0;
int ans = n / m;
 System.out.println("Answer: " + ans);
 }
}
```

OUTPUT ERROR!
Exception in thread "main" java.lang.ArithmeticException: / by zero at
Geeks.main(Main.java:9)

- The object orientation mechanism has provided the following techniques to work with:- 1. Try
2. Catch
3. Throw
4. throws
5. finally

Hierarchy of Java Exception classes:-



What is an exception to Hierarchy?

Throwable class is the super or root class of java exception hierarchy which contain two subclasses that is-

1)exception

2)error

User defined exceptions

➢ We can create our own exception by extending the exception class.

➢ The throw and throws keywords are used while implementing user defined exceptions

```
Class ownExcepion extends Exception
{
ownException(String msg)
{
Super(msg);
}
}
Class test
{
Public static void main(String args[])
Int mark=101;
Try
{
if(mark>100)
{
Throw new ownException(—Marks>100ll);
}
}
Catch(ownException e)
{
System.out.println (—Exception caughtrll); System.out.println.(—e.getMessage()); }
Finally
{
System.out.println(—End of prgll);
}
}
}
```

Output:

Exception caught

Marks is > 100

End of program

- **Try-catch block:-** try-catch block in Java is a mechanism to handle exceptions. This ensures that the application continues to run even if an error occurs. The code inside the try block is executed, and if any exception occurs, it is then caught by the catch block.

Example 1: Here, in the below example we handled the ArithmeticException using the simple try-catch block.

```
import java.io.*;
class Arithmeticexception {    public
static void main(String[] args) {       try
{
          // This will throw an ArithmeticException          int res
= 10 / 0;
       }
       // Here we are Handling the exception
catch (ArithmeticException e) {
          System.out.println("Exception caught: " + e);
   }
      // This line will executes weather an exception
   // occurs or not
      System.out.println("I will always execute");
   }
}
```

Output

Exception caught: java.lang.ArithmeticException: / by zero I will always execute

Syntax of try catch block

```
try {
 // Code that might throw an exception
} catch (ExceptionType e) {
 // Code that handles the exception
}
```

1. try in Java

The try block contains a set of statements where an exception can occur.

```
try
{
    // statement(s) that might cause exception
}
```

2. catch in Java

The catch block is used to handle the uncertain condition of a try block. A try block is always followed by a catch block, which handles the exception that occurs in the associated try block.

```
catch
{
   // statement(s) that handle an exception
   // examples, closing a connection, closing    // file, exiting the process after writing    // details to a log file.
}
```

Example 2: Here, we demonstrate the working try catch block with multiple catch statements.

```
// Java Program to Demonstrate try catch block
// with multiple catch statements
import java.util.*;

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

            // ArithmeticException
int res = 10 / 0;


        // NullPointerException
            String s = null;
            System.out.println(s.length());
        }
        catch (ArithmeticException e) {
            System.out.println(
                "Caught ArithmeticException: " + e);
        }
        catch (NullPointerException e) {
            System.out.println(
                "Caught NullPointerException: " + e);
        }
    }
}
```

Output
Caught ArithmeticException: java.lang.ArithmeticException: / by zero

❖ **Java final, finally and finalize:-** In Java, the final, finally, and finalize keywords play an important role in exception handling. The main difference between final, finally, and finalize is,

1. **final:** The "final" is the keyword that can be used for immutability and restrictions in variables, methods, and classes.
2. **finally:** The "finally block" is used in exception handling to ensure that a certain piece of code is always executed whether an exception occurs or not.
3. **finalize:** finalize is a method of the object class, used for cleanup before garbage collection.

1. final Keyword
The final keyword in Java is used with variables, methods, and also with classes to restrict modification.

Syntax:

```
// Constant value final int a
= 100;
```

Example: The below Java program demonstrates the value of the variable cannot be changed once initialized.

```
public class FinalVariableDemo {    public static
void main(String[] args) {       final int x = 10; //
Initialize final variable
       // Try to change the value of x
       // x = 20; // This will result in a compile-time error
   System.out.println("Value of x: " + x);
   }
}
```

Output:
Value of x: 10

2. finally Keyword
The finally keyword in Java is used to create a block of code that always executes after the try block, regardless of whether an exception occurs or not.

Syntax:

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code that will always execute
}
```

Example: The below Java program demonstrates the working of finally block in exception handling.

```
public class FinallyBlockDemo {     public
static void main(String[] args) {        try {
            System.out.println("Inside try block");            int num = 10 / 0; // This
will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Inside catch block");
            System.out.println("Exception caught: " + e.getMessage());
    } finally {
            System.out.println("Inside finally block");
        }
    }
}
```

Output:

```
Inside try block
Inside catch block
Exception caught: / by zero
Inside finally block
```

3. finalize() Method

The finalize() method is called by the Garbage Collector just before an object is removed from memory. It allows us to perform clean up activity. Once the finalized method completes, Garbage Collector destroys that object.finalize method is present in the Object class.

Syntax:

```
protected void finalize throws Throwable{}
```

Example: The below Java program demonstrates the working of finalize() method in context of garbage collection.

```
class Test {    @Override
   protected void finalize() throws Throwable {
   System.out.println("finalize() method called");    }

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

Output:
finalize() method called

- **Throw Keyword:-** the throw keyword is used to explicitly throw the user defined or costomized exception object to the JVM explicity for that purpose we use throw keybord

Ex:-
```
 if age(age<18)
{
Throw new invalidageexception(“can’t eligible for vote”); }
Else
{
(“eligible for vote”);
}
```

program:- Class
```
throwDemo
{
   Public static void main(String[] args){
   //System.out.println(10/10);
   Throw new ArithmeticException(“/ by zero”)
}
```

Output:-
Exception in thread “main” java.langArithmeticException:/by zero At
throwDemo,.main(throwDemo..java:7)

Throw and throws keyword:-

- **Throws keyword:-** “throws” keyword is used when we doesn't want to handle the exception and try to send the exception to the JVM(JVM or other method).

Syntax of Java throws
type method_name(parameters) throws exception_list

Example 1: Unhandled Exception

```
Class throwsDemo
{
    Public static void main(String[] args)
    {       for(int i=1;i<10;i++)
        {
            System.out.println(i);
             Thread.sleep(1000);
        }
     }
}
```

Output error

Example 2: Using throws to Handle Exception

```
class throwsDemo
{
   public static void main(String[] args)       throws
InterruptedException
   {
      for(int i=1;i<10;i++)
      {
         System.out.println(i);
              Thread.sleep(1000);
      }
   }
}
```

Output

1
2
3
4
5
6
7
8
9
10

Important Points to Remember

- throws keyword is required only for checked exceptions and usage of the throws keyword for unchecked exceptions is meaningless.
- throws keyword is required only to convince the compiler and usage of the throws keyword does not prevent abnormal termination of the program.
- With the help of the throws keyword, we can provide information to the caller of the method about the exception.

Difference between throw and throws:-

throw	**throws**
Throw keyword is used to throw an exception object explicitly.	Throw keyword is used to declare an exception as well as to bypass the caller.
Throw keyword always present inside method	Throw keyword always used with method signature
We can throw only one exception at a time.	We can only handle multiple exceptions using the throw keyword.
Throw is followed by an instance.	Throws is followed by class

CLASS AND OBJECT IN JAVA

What is a Class?
A class is a blueprint or template that defines attributes (fields) and behaviors (methods) that objects created from the class will have.

Example:

```
java
CopyEdit
public class Car {
    // Attributes (fields)
    String color;
    int speed;

    // Behavior (method)
    void drive() {
        System.out.println("The car is driving.");
    }
}
```

What is an Object?

An **object** is an instance of a class. It holds actual values for the attributes and can use the behaviors defined in the class.

Example:

java

CopyEdit

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of Car
        Car myCar = new Car();

        // Setting values
        myCar.color = "Red";
        myCar.speed = 100;

        // Accessing method
        myCar.drive();

        // Printing values
        System.out.println("Color: " + myCar.color);
        System.out.println("Speed: " + myCar.speed);
    }
}
```

Control Statements in Java

Control statements in Java are used to dictate the flow of execution of a program. They allow the program to make decisions, repeat certain actions, and jump to different parts of the code based on specific conditions. Java provides three main types of control statements: **Decision-Making Statements**, **Loop Statements**, and **Jump Statements**.

Decision-Making Statements:-

Decision-making statements evaluate a Boolean expression and control the program flow based on the result. The primary decision-making statements in Java are:

- **If Statement**: Evaluates a condition and executes a block of code if the condition is true.

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

- **If-Else Statement**: Executes one block of code if the condition is true and another block if the condition is false.

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

- **If-Else-If Ladder**: Allows multiple conditions to be checked in sequence.

```
if (condition1) {
// Statements to execute if condition1 is true
} else if (condition2) {
// Statements to execute if condition2 is true
} else {
// Statements to execute if none of the above conditions are true }
```

- **Switch Statement**: Evaluates a variable and executes a block of code based on the matching case.

```
switch (variable) {
case value1:
// Statements
break;
case value2:
// Statements
break;
default:
// Default statements
}
```

Loop Statements:

Loop statements are used to execute a block of code repeatedly based on a condition. The main loop statements in Java are:

- **For Loop**: Used when the number of iterations is known.

```
for (initialization; condition; update) {
// Statements to execute
}
```

- **While Loop**: Used when the number of iterations is not known and the condition is checked before the loop body.

```
while (condition) {
// Statements to execute
}
```

- **Do-While Loop**: Similar to the while loop, but the condition is checked after the loop body, ensuring the loop executes at least once.

```
do {
// Statements to execute
} while (condition);
```

- **For-Each Loop**: Used to iterate over elements in a collection or array. for (type element : collection) {

```
// Statements to execute
}
```

Jump Statements:

Jump statements are used to transfer control to another part of the program. The main jump statements in Java are:

- **Break Statement**: Terminates the loop or switch statement and transfers control to the statement immediately following the loop or switch.

```
for (int i = 0; i < 10; i++) {
if (i == 5) break;
// Statements
}
```

- **Continue Statement**: Skips the current iteration of a loop and proceeds to the next iteration. for (int i = 0; i < 10; i++) {

```
if (i % 2 == 0) continue;
// Statements
}
```

- **Return Statement**: Exits a function and optionally returns a value to the caller.

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

Control statements are fundamental to creating flexible and efficient Java programs, allowing developers to manage the flow of execution based on various conditions and requirements.

CONSTRUCTOR:-

- A constructor is a special member function having name same as class name.
- The constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors look a little strange because they have no return type, not even **void**.
- Its purpose is to initialize the object of a class.
- It is of three types:
  - **Default Constructor:** Constructor which takes no argument(s) is called Default Constructor.
  - **Parameterized constructor:** Constructor which takes argument(s) is called parameterized Constructor.
  - **Copy Constructor:** Constructor which takes object as its argument is called copy constructor.
- When a single program contains more than one constructor, then the constructor is said to be overloaded.

PROGRAM: WAP to illustrate constructor overloading in java.

Solution:

```
class Box
{
        double width;
        double height;
        double depth;
        Box(double w, double h, double d)          // Parameterized Constructor
        {
        width = w;
        height = h;
        depth = d;
        }

        Box()                                      // Default Constructor
        {
        width = -1;
        height = -1;
        depth = -1;
        }

        Box(double len)                            // Parameterized Constructor
        {
        width = height = depth = len;
        }

        double volume()
        {
        return width * height * depth;
        }
}

class Overload
{
        public static void main(String args[])
        {
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                double vol;
                vol = mybox1.volume();
```

```
                System.out.println("Volume of mybox1 is " + vol);
                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
        }
}
```

Output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines this keyword. this can be used inside any method to refer to the current object. **It always points to the object through which a method is called.** We can use this anywhere a reference to an object of the current class' type is permitted.

To better understand what this refers to, consider the following version of Box ( ):

```
// A redundant use of this.

Box(double w, double h, double d)
{
        this.width  =  w;
        this.height  =  h;
        this.depth = d;
}
```

This version of Box ( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box ( ), this will always refer to the invoking object.

The static Keyword

To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. We can declare both methods and variables to be static.

The most common example of a static member is main ( ). main ( ) is declared as static because it must be called before any objects exist.

WAP to demonstrate use of static variables and methods.

Solution:

```
class UseStatic
{

        static int a = 3;
        static int b;
        static void meth(int x)
        {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        }
        static
        {
        System.out.println("Static block initialized.");
        b = a * 4;
        }
        public static void main(String args[])
        {
        meth(42);
        }
}
```

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes (printing a message), and finally, b is initialized to a * 4 or 12. Then main( )is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Note: *It is illegal to refer to any instance variables inside of a static method.*

The final Keyword

A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared. (In this usage, **final** is similar to **const** in C/C++/C#.) For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
```

Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.

The keyword **final** can also be applied to methods (will be discussed in inheritance), but its meaning is substantially different than when it is applied to variables.

Garbage Collection

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.

- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation automatically. The technique that accomplishes this is called garbage collection.
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

The finalize () method

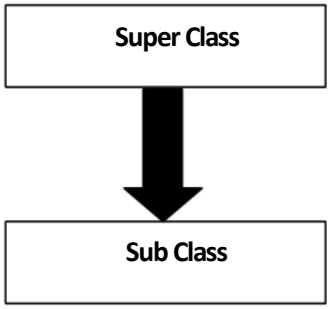
- Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- To add a finalizer to a class, we simply define the **finalize ( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize ( )** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize ( )** method on the object.

The **finalize ( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

Inheritance

- It is the process by which object of one class can acquire the properties of object of another class.
- The class, from which properties are inherited, is known as **super class**.
- The class, to which properties are inherited, is known as **sub class**. A sub class inherits all of the variables and methods defined by the super class and add its own, unique elements.
- Inheritance allows the creation of hierarchical classifications.



- To derive a sub class from super class, we use the **extends** keyword.
- The general form of a class declaration that inherits a superclass is :

```
class subclass-name extends superclass-name
{
 // body of class
}
```

- A sub class can access all the public and protected members of a super class. (By default, the variable or function of a class are public in nature)

How Constructor method of a Super class gets called

- A subclass constructor invokes the constructor of the super class implicitly.
  - When a BoxWeight object, a subclass, is instantiated, the default constructor of its super class, Box class, is invoked implicitly before sub-class's constructor method is invoked.
- A subclass constructor can invoke the constructor of the super explicitly by using the "super" keyword.
  - The constructor of the BoxWeight class can explicitly invoke the constructor of the Box class using "super" keyword.
- In a class hierarchy, constructors are called in order of derivation, from super class to subclass.
- Since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used.

If **super( )** is not used, then the default of each super class will be executed

Package and Interface

- Packagesarecontainersforclassesthatareusedtokeeptheclass namespace compartmentalized.
- For example, a package allows us to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

Defining a Package

- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.)
- While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.
- This is the general form of the **package** statement: package pkg;

  Here, pkg is the name of the package. For example, the following statement creates a package called **MyPackage**. package MyPackage;

package
MyPack; class
Balance

```
{
        String
        name;
        double bal;
        Balance(String n, double b)
        {
        name = n;
        bal = b;
        }
        void show()
        {
        if(bal<0)
        System.out.print("-->
        ");
        System.out.println(name + ": $" + bal);
        }
        }
        class AccountBalance
        {
        public static void main(String args[])
        {
        Balance current[] = new Balance[3];

        current[0] = new Balance("Sanjib", 123.23);
        current[1] = new Balance("Chandan", 157.02);
        current[2] = new Balance("Palak", -12.33);

        for(int i=0; i<3; i++)
        current[i].show();
        }
}
```

Call this file **AccountBalance.java**, and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then try executing the **AccountBalance** class, using the following command line:

java MyPack.AccountBalance

Remember, you will need to be in the directory above **MyPack** when you execute this command, or to have your **CLASSPATH** environmental variable set appropriately. As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

java AccountBalance

AccountBalance must be qualified with its package name.

Access Protection

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Importing Packages

- Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages.
- In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement: import pkg1 [.pkg2].(classname|*);

```
package MyPack;
public class Balance
{
        String
        name;
        double bal;
        public Balance(String n, double b)
        {
        name = n;
        bal = b;
        }
        public void show()
        {
        if(bal<o)
        System.out.print("-->
        ");
        System.out.println(name + ": $" + bal);
        }
}
```

As we can see, the **Balance** class is now **public**. Also, its constructor and its **show( )** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import
MyPack.*; class
TestBalance
{
        public static void main(String args[])
        {
        Balance test = new Balance("J. J. Jaspers",
        99.88); test.show(); // you may also call show()
        }
}
```

Interface

- An interface is a collection of abstract methods (i.e. methods without having definition).
- A class that implements an interface inherits abstract methods of the interface.
- An interface is not a class.
- Writing an interface is similar to writing a class, but they differ in concepts.
- A class describes both attributes (i.e. variables) and behaviors (i.e. methods) of an object.
- An interface contains only behaviors (i.e. methods) that a class implements.
- If a class (**provided it is not abstract**) implements an interface, then all the methods of interface need to be defined in it.

Declaring Interface

The **interface** keyword is used to declare an interface. Here is a simple example to declare an

interface: public interface NameOfInterface

```
{
  //Any number of final, static variables
  //Any number of abstract method declarations (i.e. method without definitions)
}
```

Interfaces have the following properties:

- An interface is implicitly (i.e. by default) abstract. We do not need to use the **abstract** keyword when declaring an interface.
- Each method in an interface is also implicitly (i.e. by default) abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly (i.e. by default) public.

Implementing Interface:

- A class can implement more than one interface at a time.
- A class can extend only one class, but can implement many interfaces.
- An interface itself can extend another interface.
- The general form of a class that includes the **implements** clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]]
{
// class-body
}
```

Example:

```
interface Message
{
void
message1();
void
message2();
}

class A implements Message
{
        void message1()
        {
        System.out.println("Good Morning");
        }

        void message2()
        {
        System.out.println("Good Evening");
        }

        public static void main(String args[])
        {
        A a=new A();
        a.message1();
        a.message2();
        }
}
```

Similarities between class and iinterface

- Both class and interface can contain any number of methods.
- Both class and interface are written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The bytecode of both class and interface appears in a **.class** file.

Dissimilarities between class and interfcae

- We cannot instantiate (i.e. create object) an interface but we can instantiate a class.
- An interface does not contain any constructors but a class may contain any constructors.
- All the methods in an interface are abstract (i.e. without definitions) but in a class method may or may not be abstract.
- An interface cannot contain variables. The only variable that can appear in an interface must be declared both static and final. But a class can contain any variable.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces (i.e. multiple inheritances can be achieved through it). But a class can not extend multiple classes (i.e multiple inheritance can not be achieved).

Variables and Interface:

- We can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values.
- When we include that interface in a class (that is, when we “implement” the interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations.
- If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything.

```
interface SharedConstants
{
int X = 0;
int Y = 1;
}
class A implements SharedConstants
{
        static void show()
        {
        System.out.println("X="+x+"and Y="+y);
        }

        public static void main(String args[])
        {
        A a = new A();
        show();
        }
}
```

Interfaces Can Be Extended:-

- One interface can inherit another by use of the keyword **extends**.
- The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
interface A
{
void meth1();
void meth2();
}

interface B extends A
{
void meth3();
}

class MyClass implements B
{
        public void meth1()
        {
```

```
System.out.println("Implement meth1().");
```

```
        }
        public void meth2()
        {
        System.out.println("Implement meth2().");
        }
        public void meth3()
        {
        System.out.println("Implement meth3().");
        }
}
class IFExtend
{
        public static void main(String arg[])
        {
        MyClass ob = new MyClass();
        ob.meth1();
        ob.meth2();
        ob.meth3();
        }
}
```

Multithreading in Java

Multithreading is a Java feature that allows the concurrent execution of two or more parts of a program for maximum utilization of the CPU. Each part of such a program is called a thread. So, threads are lightweight processes within a process.

Different Ways to Create Threads

Threads can be created by using two mechanisms:

1.Extending the Thread class

2.Implementing the Runnable Interface

---

1. By Extending the Thread Class

We create a class that extends the **java.lang.Thread** class. This class overrides the **run()** method available in the Thread class. A thread begins its life inside the run() method. We create an object of our new class and call the **start()** method to start the execution of a **thread.start()** invokes the run() method on the Thread object.

Example: Creating a Thread by extending the Thread class

```
// Java code for thread creation by extending
// the Thread class
class Multithreading extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
            + " is running");
        }
        catch (Exception e) {

            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Geeks
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
```

```
        Multithreading object
            = new Multithreading();
        object.start();
    }
  }
}
```

Output:

```
Thread 13 is running
Thread 14 is running
Thread 12 is running
Thread 11 is running
Thread 16 is running
Thread 15 is running
Thread 18 is running
Thread 17 is running
```

2.By Implementing the Runnable Interface

We create a new class which implements **java.lang.Runnable** interface and override **run()** method. Then we instantiate a Thread object and call start() method on this object.

Example: Demonstrating the multithreading by implementing the Runnable interface.

```
// Java code for thread creation by implementing // the
Runnable Interface
class Multithreading implements Runnable {
  public void run()
  {
    try {
      // Displaying the thread that is running
      System.out.println(
        "Thread " + Thread.currentThread().getId()
      + " is running");
    }
    catch (Exception e) {

      // Throwing an exception
      System.out.println("Exception is caught");
    }
  }
}

// Main Class
class Geeks {
  public static void main(String[] args)
```

```
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            Thread object
                = new Thread(new Multithreading());
            object.start();
        }
    }
}
```

Output

Thread 12 is running

Thread 15 is running

Thread 18 is running

Thread 11 is running

Thread 17 is running

Thread 13 is running

Thread 16 is running

Thread 14 is running

Lifecycle and States of a Thread in Java

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:

1.New State

2.Runnable State

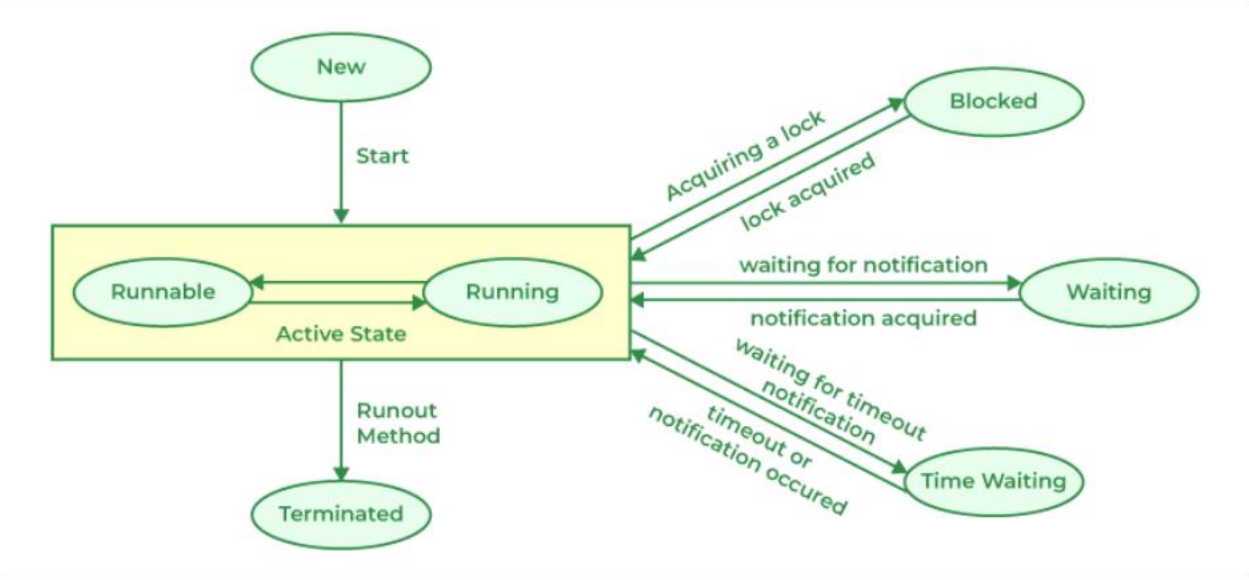
3.Blocked State

4.Waiting State

5.Timed Waiting State

6.Terminated State

The diagram below represents various states of a thread at any instant:



Life Cycle of a Thread

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

1. **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 has not started to execute.

2. **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.

3. **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.

4. **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.

5. **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.

6. **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.

Thread States in Java

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread. Java provides **java.lang.Thread.State** enum that defines the ENUM constants for the state of a thread, as a summary of which is given below:

1. New

Thread state for a thread that has not yet started.

public static final Thread.State NEW

2. Runnable

Thread state for a runnable thread. A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as a processor.

public static final Thread.State RUNNABLE

3. Blocked

Thread state for a thread blocked waiting for a monitor lock. A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling **Object.wait().**

public static final Thread.State BLOCKED

4. Waiting

Thread state for a waiting thread. A thread is in the waiting state due to calling one of the following methods:

- Object.wait with no timeout
- Thread.join with no timeout
- LockSupport.park

public static final Thread.State WAITING

5. Timed Waiting

Thread state for a waiting thread with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:

- Thread.sleep
- Object.wait with timeout
- Thread.join with timeout
- LockSupport.parkNanos

- LockSupport.parkUntil

public static final Thread.State TIMED_WAITING

6. Terminated

Thread state for a terminated thread. The thread has completed execution.

public static final Thread.State TERMINATED

Example of Demonstrating Thread States

Below is a real-world example of a ticket booking system that demonstrates different thread states:

Example:

```
// Java program to demonstrate thread states
// using a ticket booking scenario
classTicketBookingimplements Runnable {
    @Override
    public void run() {

        try {

            // Timed waiting
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("State of bookingThread while mainThread is waiting: " +
                TicketSystem.mainThread.getState());

        try {

            // Another timed waiting
            Thread.sleep(100);
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

publicclassTicketSystemimplements Runnable {
    publicstatic Thread mainThread;
    publicstatic TicketSystem ticketSystem;

    @Override
    public void run() {
        TicketBooking booking = new TicketBooking();
        Thread bookingThread = new Thread(booking);

        System.out.println("State after creating bookingThread: " + bookingThread.getState());

        bookingThread.start();
        System.out.println("State after starting bookingThread: " + bookingThread.getState());

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("State after sleeping bookingThread: " + bookingThread.getState());

        try {
```

```
            // Moves mainThread to waiting state
            bookingThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("State after bookingThread finishes: " + bookingThread.getState());
    }

    public static void main(String[] args) {
        ticketSystem = new TicketSystem();
        mainThread = new Thread(ticketSystem);

        System.out.println("State after creating mainThread: " + mainThread.getState());

        mainThread.start();
        System.out.println("State after starting mainThread: " + mainThread.getState());
    }
}
```

Output:

```
C:\Users\GFG19656\Documents\Java Thread>java TicketSystem
State after creating mainThread: NEW
State after starting mainThread: RUNNABLE
State after creating bookingThread: NEW
State after starting bookingThread: RUNNABLE
State after sleeping bookingThread: TIMED_WAITING
State of bookingThread while mainThread is waiting: WAITING
State after bookingThread finishes: TERMINATED
```

Explanation:

- When a new thread is created, the thread is in the NEW state. When the start() method is called on a thread, the thread scheduler moves it to the Runnable state.

- Whenever the join() method is called on a thread instance, the main thread goes to Waiting for the booking thread to complete.
- Once the thread's run method completes, its state becomes Terminated.

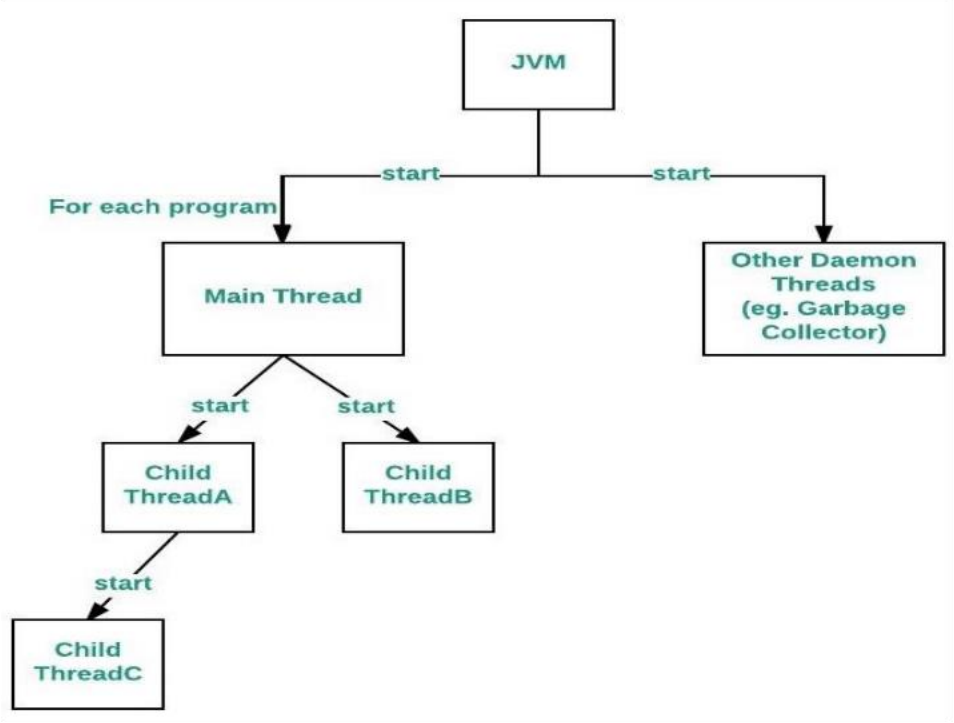
Main thread in Java

Java provides built-in support for multithreaded programming. A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.
When a Java program starts up, one thread begins running immediately. This is usually called the *main* thread of our program because it is the one that is executed when our program begins.

There are certain properties associated with the main thread which are as follows:

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions

The flow diagram is as follows:



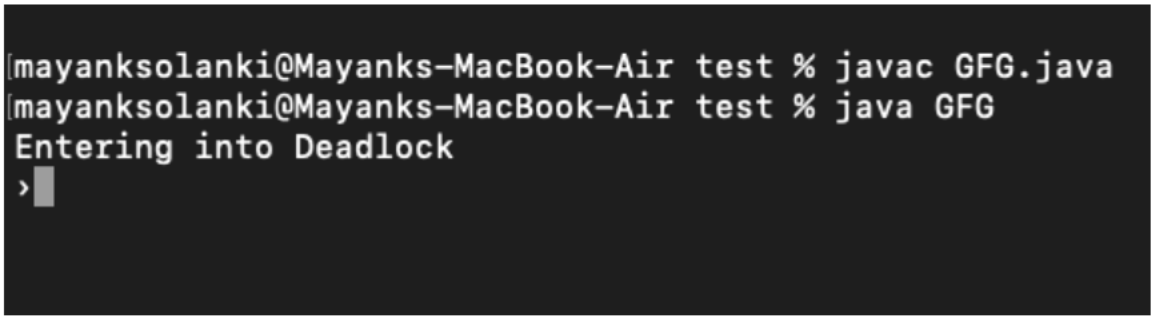
Example:

// Java program to demonstrate deadlock

// using Main thread

```
// Main class
public class GFG {

// Main driver method
public static void main(String[] args) {
        // Try block to check for exceptions
        try {
        // Print statement
        System.out.println("Entering into Deadlock");
        // Joining the current thread
        Thread.currentThread().join();
        // This statement will never execute
        System.out.println("This statement will never execute");
        }
        // Catch block to handle the exceptions
        catch (InterruptedException e) {
        // Display the exception along with line number
        // using printStackTrace() method
        e.printStackTrace();
        }
}
}
```

Output



Priority Levels in Java

Java provides three priority constants in the Thread class:

java

CopyEdit

```
Thread.MIN_PRIORITY = 1;
Thread.NORM_PRIORITY = 5; // default
Thread.MAX_PRIORITY = 10;
```

You can set the priority of a thread using:

java

CopyEdit

```
thread.setPriority(int priority);
```

You can retrieve it using:

java

CopyEdit

```
int priority = thread.getPriority();
```

Example

java

CopyEdit

```
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " with priority " +
Thread.currentThread().getPriority());
    }
}
public class ThreadPriorityExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
```

```
        MyThread t3 = new MyThread();
        t1.setPriority(Thread.MIN_PRIORITY); // 1
        t2.setPriority(Thread.NORM_PRIORITY); // 5
        t3.setPriority(Thread.MAX_PRIORITY); // 10
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output may vary, but in some JVMs you might see the higher-priority thread getting scheduled more often.

Key Points

- **Default priority** is 5.
- **Priorities are just hints** to the JVM scheduler.
- **Thread behavior is platform-dependent**; don't rely on priority for program correctness.
- Instead of priorities, consider using ExecutorService for better thread management.

Synchronization in Java

In Multithreading, **Synchronization** is crucial for ensuring that multiple threads operate safely on shared resources. Without **Synchronization**, data inconsistency or corruption can occur when multiple threads try to access and modify shared variables simultaneously. In Java, it is a mechanism that ensures that only one thread can access a resource at any given time. This process helps prevent issues such as data inconsistency and race conditions when multiple threads interact with shared resources.

Example: Below is the Java Program to demonstrate synchronization.

```
// Java Program to demonstrate synchronization in Java
class Counter {
    private int c = 0; // Shared variable

    // Synchronized method to increment counter
    public synchronized void inc() {
        c++;
    }

    // Synchronized method to get counter value
```

```
    public synchronized int get() {
        return c;
    }
}

public class Geeks {
    public static void main(String[] args) {
        Counter cnt = new Counter(); // Shared resource

        // Thread 1 to increment counter
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                cnt.inc();
            }
        });

        // Thread 2 to increment counter
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                cnt.inc();
            }
        });

        // Start both threads
        t1.start();
        t2.start();

        // Wait for threads to finish
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Print final counter value
        System.out.println("Counter: " + cnt.get());
    }
}
```

Output

Counter: 2000

Synchronized Blocks in Java:

Java provides a way to create threads and synchronize their tasks using **synchronized blocks.**

A **synchronized block** in Java is synchronized on some object. Synchronized blocks in Java are marked with the **synchronized** keyword. All synchronized blocks synchronize on the same object and can only have one thread executed inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block. If you want to master concurrency and understand how to avoid common pitfalls,

General Form of Synchronized Block

```
synchronized(sync_object)
{
// Access shared variables and other
// shared resources
}
```

Example: Below is an example of synchronization using Synchronized Blocks

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

    public void increment() {
        synchronized (this) {
            count++;
        }
    }
       public int getCount() {
        synchronized (this) {
            return count;
        }
    }
}
class CounterThread extends Thread {
    private Counter counter;

    public CounterThread(Counter counter) {
        this.counter = counter;
```

```
    }
    public void run() {
        for (int i = 0; i < 10000; i++) {
            counter.increment();
        }
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        CounterThread thread1 = new CounterThread(counter);
        CounterThread thread2 = new CounterThread(counter);

        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println("Final count: " + counter.getCount());
    }
}
```

2. Thread Synchronization in Java

Thread Synchronization is used to coordinate and ordering of the execution of the threads in a multi-threaded program. There are two types of thread synchronization are mentioned below:

- Mutual Exclusive
- Cooperation (Inter-thread communication in Java)

Example: Java Program to demonstrate thread synchronization for Ticket Booking System

```
// Java Program to demonstrate thread synchronization for Ticket Booking System
classTicketBooking {
    private int availableTickets = 10; // Shared resource (available tickets)
```

```
    // Synchronized method for booking tickets
    publicsynchronized void bookTicket(int tickets) {
        if (availableTickets >= tickets) {
            availableTickets -= tickets;
            System.out.println("Booked " + tickets + " tickets, Remaining tickets: " + availableTickets);
        } else {
            System.out.println("Not enough tickets available to book " + tickets);
        }
    }

    public int getAvailableTickets() {
        return availableTickets;
    }
}

publicclassGeeks {
    publicstatic void main(String[] args) {
        TicketBooking booking = new TicketBooking(); // Shared resource

        // Thread 1 to book tickets
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 2; i++) {
                booking.bookTicket(2); // Trying to book 2 tickets each time
                try {
                    Thread.sleep(50); // Simulate delay
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
```

```
        // Thread 2 to book tickets
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 2; i++) {
                booking.bookTicket(3); // Trying to book 3 tickets each time
                try {
                    Thread.sleep(40); // Simulate delay
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        // Start both threads
        t1.start();
        t2.start();

        // Wait for threads to finish
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Print final remaining tickets
        System.out.println("Final Available Tickets: " + booking.getAvailableTickets());
    }
}
```

Output

Booked 2 tickets, Remaining tickets: 8

Booked 3 tickets, Remaining tickets: 5

Booked 3 tickets, Remaining tickets: 2

Booked 2 tickets, Remaining tickets: 0

Final Available Tickets: 0

Explanation: Here the TicketBooking class contains a **synchronized** method **bookTicket()**, which ensures that only one thread can book tickets at a time, preventing race conditions and overbooking. Each thread attempts to book a set number of tickets in a loop, with thread synchronization ensuring that the availableTickets variable is safely accessed and updated. Finally, the program prints the remaining tickets.

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. There are three types of Mutual Exclusive mentioned below:

- Synchronized method.
- Synchronized block.
- Static synchronization.

Example: Below is the implementation of the Java Synchronization

```
// A Java program to demonstrate working of synchronized.

importjava.io.*;

// A Class used to send a message

classSender {

   public void send(String msg)

   {

      System.out.println("Sending " + msg);  // Changed to print without new line

      try {

         Thread.sleep(100);

      }

      catch (Exception e) {

         System.out.println("Thread  interrupted.");

      }
```

```
        System.out.println(msg + "Sent"); // Improved output format
    }
}

// Class for sending a message using Threads
class ThreadedSend extends Thread {
    private String msg;
    Sender sender;

    // Receives a message object and a string message to be sent
    ThreadedSend(String m, Sender obj)
    {
        msg = m;
        sender = obj;
    }

    public void run()
    {
        // Only one thread can send a message at a time.
        synchronized (sender)
        {
            // Synchronizing the send object
            sender.send(msg);
        }
    }
}

// Driver class
class Geeks {
    public static void main(String args[])
```

```
  {
      Sender send = new Sender();
      ThreadedSend S1 = new ThreadedSend("Hi ", send);
      ThreadedSend S2 = new ThreadedSend("Bye ", send);

      // Start two threads of ThreadedSend type
      S1.start();
      S2.start();

      // Wait for threads to end
      try {
         S1.join();
         S2.join();
      }
      catch (Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

Output

Sending Hi

Hi Sent

Sending Bye

Bye Sent

Explanation: In the above example, we choose to synchronize the Sender object inside the run() method of the ThreadedSend class. Alternately, we could define the **whole send() block as synchronized**,producing the same result. Then we don't have to synchronize the Message object inside the run() method in the ThreadedSend class.

We do not always have to synchronize a whole method. Sometimes it is preferable to **synchronize only part of a method**. Java synchronized blocks inside methods make this possible.

Java I/O

- A stream is continuous group of data or a channel through which data flows from one point to another point.
- Java implements streams within class hierarchies defined in the **java.io** package.
- **Byte streams** provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.
- **Character streams** provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

Stream

Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understands the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the **java.io** package. Java 2 defines two types of streams: **byte** and **character.**

Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. **Character streams** provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The Byte Stream Classes

- Byte streams are defined by using two class hierarchies. At the top are two abstract classes:

  InputStream and OutputStream.

- To use the stream classes, we must import **java.io**.
- The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement.
- Two of the most important are **read ( )** and **write ( )**, which, respectively, read and write bytes of data. Both methods are declared as abstract inside **InputStream** and **OutputStream**. They are overridden by derived stream classes.

The Character Stream Classes

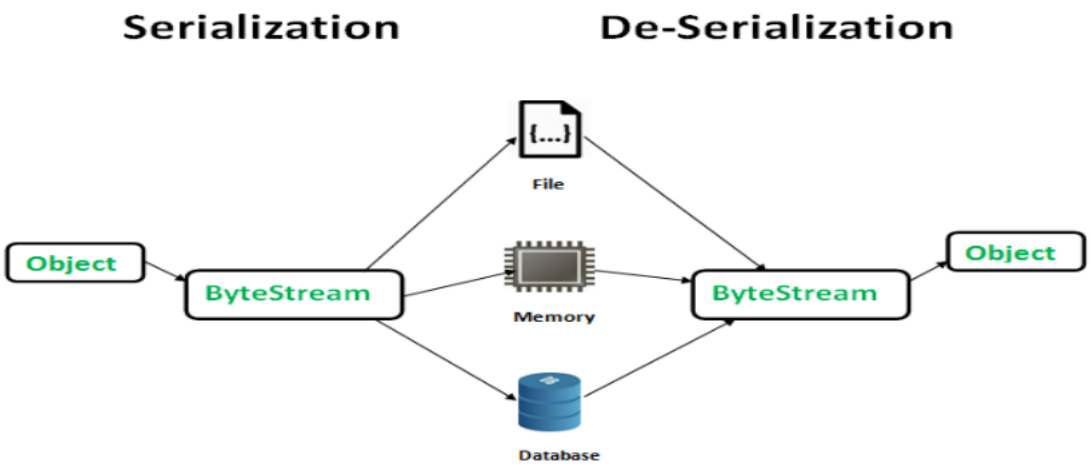
- Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these.
- The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read ( )** and write **( )**, which read and write characters of data, respectively. These methods are overridden by derived stream classes.

Java System Class

- All Java programs automatically import the **java.lang** package.
- This package defines a class called **System**, which encapsulates several aspects of the run-time environment.
- **System** also contains three predefined stream variables, **in**, **out**, and **err**. These fields are declared as **public** and **static** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object.
- **System.out** refers to the standard output stream. By default, this is the console.
- **System.in** refers to standard input, which is the keyboard by default.

Serialization:-

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the **java.io.Serializable** interface. The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

```
public final void writeObject(Object obj)
                    throws IOException
```

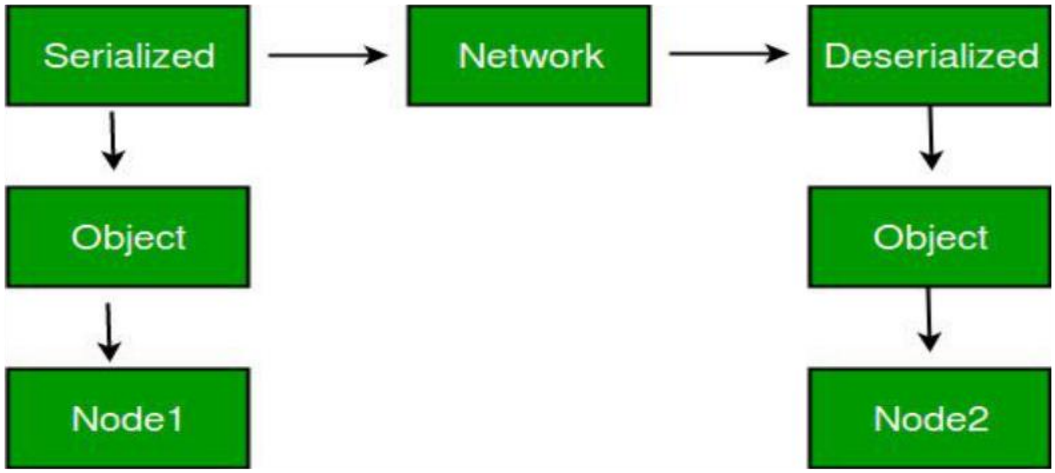
The ObjectInputStream class contains **readObject()** method for deserializing an object.

```
public final Object readObject()
                    throws IOException,
                ClassNotFoundException
```

Advantages of Serialization

1.To save/persist state of an object.

2.To travel an object across a network.



Only the objects of those classes can be serialized which are implementing **java.io.Serializable** interface. Serializable is a **marker interface** (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

Points to remember

1. If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice-versa is not true.
2. Only non-static data members are saved via Serialization process.
3. Static data members and transient data members are not saved via Serialization process. So, if you don't want to save value of a non-static data member then make it transient.
4. Constructor of object is never called when an object is deserialized.
5. Associated objects must be implementing Serializable interface. Example :

```
class A implements Serializable{
// B also implements Serializable
// interface.
B ob=new B();
}
```

Example

Demonstration of object serialization.

Solution:

```
import java.io.*;
class Employee implements java.io.Serializable
{
  public String name;
  public String address;
  public transient int age;
  public int number;
  public void mailCheck()
  {
    System.out.println("Mailing a check to " + name + " " + address);
  }
}


public class SerializeDemo
{
  public static void main(String [] args)
  {
                Employee e = new Employee();
                e.name = "Sourav Kumar Giri";
                e.address = "Baleswar, Odisha.";
                e.age =25;
                e.number = 101;
                try
                {
                  FileOutputStream fileOut =new FileOutputStream("employee.ser");
                  ObjectOutputStream out =new ObjectOutputStream(fileOut);
                  out.writeObject(e);
                  out.close();
                  fileOut.close();
                }
                catch(IOException i)
                {
                   i.printStackTrace();
                }
  }
}
```

Deserialization in java

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output:

Program: Demonstration of object de-serialization.

Solution:

```
import java.io.*;
public class DeserializeDemo
{
  public static void main(String [] args)
  {
        Employee e = null;
        try
        {
          FileInputStream fileIn =new FileInputStream("employee.ser");
          ObjectInputStream in = new ObjectInputStream(fileIn);
          e = (Employee) in.readObject();
          in.close();
          fileIn.close();
        }
        catch(IOException i)
        {
          i.printStackTrace();
          return;
        }
        catch(ClassNotFoundException c)
        {
          System.out.println("Employee class not found");
          c.printStackTrace();
          return;
        }
        System.out.println("Deserialized Employee...");
        System.out.println("Name: " + e.name);
        System.out.println("Address: " + e.address);
        System.out.println("Age " + e.age);
        System.out.println("Number: " + e.number); }
```

}

Output: Deserialized Employee...
Name: Sourav Kumar Giri
Address: Baleswar, Odisha
Age: 0
Number:101

Here are following important points to be noted:

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of readObject() is cast to an Employee reference.
- The value of the age field was 25 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The age field of the deserialized Employee object is 0.

URL:

In Java, URLConnection is part of the java.net package and allows you to connect to a URL and interact with the resource (like reading a webpage or sending data to a server).

Basic Steps to Use URLConnection

1. Create a URL object.
2. Call openConnection() on the URL.
3. Configure the connection (optional).
4. Connect and read/write data.

Example: Reading from a URL (GET request) java

```
CopyEdit
import java.io.*;
import java.net.*;

public class URLConnectionExample {
   public static void main(String[] args) {
      try {
         URL url = new URL("https://www.example.com");
   URLConnection connection = url.openConnection();
   BufferedReader in = new BufferedReader(new
InputStreamReader(connection.getInputStream()));

         String inputLine;
         while ((inputLine = in.readLine()) != null) {
            System.out.println(inputLine);
         }
```

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

Java Applet Basics

Java Applets was once a very popular feature of web applications. Java Applets were small programs written in Java that ran inside a web browser. Learning about Applet helps us understand how Java has evolved and how it handles graphics.

Note: **java.applet package** has been deprecated in Java 9 and later versions, as applets are no longer widely used on the web.

Java Applets

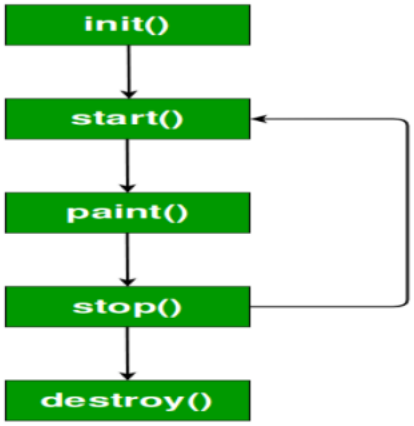
A Java Applet is a Java program that runs inside a web browser. An Applet is embedded in an HTML file using <applet> or <objects> tags. Applets are used to make the website more dynamic and entertaining. Applets are executed in a sandbox for security, restricting access to local system resources.

Key Points:

- **Applet Basics**: Every applet is a child/subclass of the java.applet.Applet class.
- **Not Standalone**: Applets don't run on their own like regular Java programs. They need a web browser or a special tool called the applet viewer (which comes with Java).
- **No main() Method**: Applets don't start with main() method.
- **Display Output:** Applets don't use System.out.prinln() for displaying the output, instead they use graphics methods like drawString() from the AWT (Abstract Window ToolKit).

Java Applet Life Cycle

The below diagram demonstrates the life cycle of Java Applet:



It is important to understand the order in which the various methods shown in the above image are called.

- When an applet begins, the following methods are called, in this sequence:
  - init( )
    - start( )
    - paint( )
- When an applet is terminated, the following sequence of method calls takes place:
  - stop( )
    - destroy( )
    -

Let's look more closely at these methods.

1. init( ): The init( ) method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

2. start( ): The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. ***Note****: init( ) is called once i.e. when the first time an applet is loaded whereas start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( )*

3. paint( ): The paint( ) method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.

- paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called.
- The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
- paint() is the only method among all the methods mention above (which is parameterized). This method is crucial for updating or redrawing the visual content of the applet.

Example:

```
public void paint(Graphics g)
{
// Drawing a string on the applet window
// g is an object reference of class Graphic.
g.drawString("Hello, Applet!", 50, 50);
}
```

4. stop( ): The stop( ) method is called when a web browser leaves the HTML document containing the applet, when it goes to another page.
For example: When stop( ) is called, the applet is probably running. You should use stop( ) to suspend threads that don't need to run when the applet is not visible. You can restart them when start( ) is called if the user returns to the page.

5. destroy( ): The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy( ).

Key Packages for Java Applets

- java.applet.Applet: Base class for applets.
- java.awt.Graphics: Used for drawing on the applet screen.
- java.awt: Provides GUI components and event-handling mechanisms.

Creating Hello World Applet

Let's begin with the HelloWorld applet :

```
importjava.applet.Applet;
importjava.awt.Graphics;

// HelloWorld class extends Applet
publicclassHelloWorldextends Applet {

  // Overriding paint() method
  @Override public void paint(Graphics g)
  {
    g.drawString("Hello World", 20, 20);
  }
}
```

The HTML APPLET Tag

The APPLET tag is used to start an applet from both an HTML document and from an applet viewer. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers like Netscape Navigator, Internet Explorer, and Hot Java will allow many applets on a single page.

The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

```
< APPLET [CODEBASE = codebaseURL]  CODE = appletFile  [ALT = alternateText] [NAME
= appletInstanceName]  WIDTH = pixels  HEIGHT = pixels  [ALIGN = alignment]
[VSPACE = pixels]  [HSPACE = pixels] >

[< PARAM NAME = AttributeName VALUE = AttributeValue>] [<
PARAM NAME = AttributeName2 VALUE =
AttributeValue>].....

[HTML Displayed in the absence of Java]

</APPLET>
```

CODEBASE: CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag).

NAME: NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them.

WIDTH AND HEIGHT: WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN: ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE AND HSPACE: These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME AND VALUE: The PARAM tag allows you to specify applet specific arguments in an HTML page. Applets access their attributes with the **getParameter ( )** method.

Passing Parameters to Applets

The APPLET tag in HTML allows us to pass parameters to an applet. To retrieve a parameter, the **getParameter( )** method is used. It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **Boolean** values, we need to convert their string representations into their internal formats. Here is an example that demonstrates passing parameters:

Program Demonstration of passing parameters to applet.

Solution:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet
{
        String fontName;
        int fontSize;
```

```
float leading;
boolean active;

public void start()
{
        String param;
        fontName = getParameter("fontName");
        if(fontName == null)
        fontName = "Not Found";

        param = getParameter("fontSize");
        try
        {
                if(param != null)
                fontSize = Integer.parseInt(param);
                else
                fontSize = 0;
        }
        catch(NumberFormatException e)
        {
                fontSize = -1;
        }

        param = getParameter("leading");
        try
        {
                if(param != null) // if not found
                leading = Float.valueOf(param).floatValue();
                else
                leading = 0;
        }
        catch(NumberFormatException e)
        {
                leading = -1;
        }

        param = getParameter("accountEnabled");
        if(param != null)
        active = Boolean.valueOf(param).booleanValue(); }
```

```
.
public void paint(Graphics g)
{
        g.drawString("Font name: " + fontName, 0, 10);
        g.drawString("Font size: " + fontSize, 0, 26);
        g.drawString("Leading: " + leading, 0, 42);
        g.drawString("Account Active: " + active, 0, 58); }
}
```

AppletContext and showDocument( )

One application of Java is to use active images and animation to provide a graphical means of navigating the Web that is more interesting than the underlined blue words used by hypertext. To allow your applet to transfer control to another URL, you must use the **showDocument( )** method defined by the **AppletContext** interface.

AppletContext is an interface that lets you get information from the applet's execution environment. The methods defined by AppletContext are shown in Table 19-2. The context of the currently executing applet is obtained by a call to the getAppletContext( ) method defined by Applet. Within an applet, once you have obtained the applet's context, you can bring another document into view by calling showDocument( ). This method has no return value and throws no exception if it fails, so use it carefully. There are two showDocument( ) methods.

- The method showDocument(URL) displays the document at the specified URL.
- The method showDocument(URL, where) displays the specified document at the specified location within the browser window. Valid arguments for where are "_self" (show in current frame), "_parent" (show in parent frame), "_top" (show in topmost frame), and "_blank" (show in new browser window).

Demonstration of applet context and showdocument.

Solution:

```
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="ACDemo" swidth=300 height=50>
</applet>
*/
public class ACDemo extends Applet
```

```
{
        public void start()
        {
        AppletContext ac = getAppletContext();
        URL url = getCodeBase(); // get url of this
                applet try
                {
                ac.showDocument(new URL(url+"Test.html"));
                }
                catch(MalformedURLException e)
                {
                showStatus("URL not found");
                }
        }
}
```

SOCKET PROGRAMMING

Introduction to Socket Programming

Socket programming is a way to enable communication between two or more devices over a network using a set of programming interfaces. In Java, sockets provide the mechanism to establish a connection between a client and a server, allowing them to exchange data. This is the foundation for network-based applications such as web browsers, online games, chat applications, and file transfer utilities.

Sockets act as endpoints for sending and receiving data. Java supports both **TCP (Transmission Control Protocol)** for reliable, connection-oriented communication and **UDP (User Datagram Protocol)** for faster, connectionless communication.

Note: *A "socket" is an endpoint for sending and receiving data across a network.*

Usage of Socket Programming

Socket programming is used in a wide range of real-time and networked applications, including:

- **Client-Server Applications:** Like online banking, booking systems, and email services.
- **Chat Applications:** Real-time text or voice communication between users.
- **File Transfer:** Sending and receiving files over a network.
- **Web Servers and Browsers:** HTTP communication between web clients and servers.
- **IoT Devices:** Sending data between sensors and servers.
- **Multiplayer Games:** Real-time gaming where data needs to be exchanged between players instantly.

Types of Socket

A network that is connected with two devices as a link to execute two-way communication on the network. It receives and sends data to the devices. The socket address is a combination of IP address and port. In the TCP/IP layer, a socket is bound as a port number which can identify whether the data is to be sent to an application or not. The transport layer in the socket is the core mechanism for managing and establishing communication between the devices.

Sockets are used as a communication device or interaction between the client and server. It receives information from the client and sends information to the client and disconnects it after receiving the data.

Types of Socket:

1. Datagram Sockets: Datagram sockets allow processes to use the User Datagram Protocol (UDP). It is a two-way flow of communication or messages. It can receive messages in a different order from the sending way and also can receive duplicate messages. These sockets are preserved with their boundaries. The socket type of datagram socket is SOCK_DGRAM.

Key Java Classes:

- ServerSocket – Listens for incoming client connections on the server.
- Socket – Connects to the server and facilitates communication.

Features:

- Reliable data transfer
- Error-checking and correction
- Maintains connection state

Use Cases:

- Web servers and clients
- Email communication
- Remote login (SSH)

TCP Example: Chat between Client and Server

Server Code:

```
import java.io.*;
import java.net.*;

public class TCPServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(5000);
        System.out.println("Server started... Waiting for client...");

        Socket socket = serverSocket.accept();
        System.out.println("Client connected.");

        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

        out.println("Hello from Server!");
        String message = in.readLine();
        System.out.println("Client says: " + message);

        socket.close();
```

```
        serverSocket.close();
    }
}
```

Client Code:

```
import java.io.*;
import java.net.*;

public class TCPClient {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("localhost", 5000);

                BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

        System.out.println("Server says: " + in.readLine());
        out.println("Hello Server!");

        socket.close();
    }
}
```

2. Stream Sockets: Stream socket allows processes to use the Transfer Control Protocol (TCP) for communication. A stream socket provides a sequenced, constant or reliable, and two-way (bidirectional) flow of data. After the establishment of connection, data can be read and written to these sockets in a byte stream. The socket type of stream socket is SOCK_STREAM.

Key Java Classes:

- DatagramSocket – Used to send and receive datagram packets.
- DatagramPacket – Represents the actual packet of data sent or received.

Features:

- Faster than TCP
- No connection overhead
- No guarantee of delivery or order

Use Cases:

- Video and voice streaming
- Online multiplayer games
- DNS (Domain Name System)

Example:

```
// Server side
DatagramSocket serverSocket = new DatagramSocket(9876);
byte[] receiveData = new byte[1024];
DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
serverSocket.receive(receivePacket);
// Process received data

// Client side
DatagramSocket clientSocket = new DatagramSocket();
byte[] sendData = "Hello, Server".getBytes();
InetAddress serverAddress = InetAddress.getByName("localhost");
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, serverAddress, 9876);
clientSocket.send(sendPacket);
```

Key classes:

1. **Socket:** Represents a connection between two devices.
2. **ServerSocket:** Listens for incoming socket connections.

1. Establish a Socket Connection

To connect to another machine we need a socket connection. A socket connection means both machines know each other's **IP address** and **TCP port**. The **java.net.Socket** class is used to create a **socket.**

Socket socket = new Socket("127.0.0.1", 5000)

- **The first argument**: The **IP address of Server** i.e. 127.0.0.1 is the IP address of localhost, where code will run on the single stand-alone machine.
- **The second argument**: The **TCP Port number** (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

2. Communication

To exchange data over a socket connection, **streams** are used for input and output:

- **Input Stream**: Reads data coming from the socket.
- **Output Stream**: Sends data through the socket.

Example to access these streams:

// to read data

InputStream input = socket.getInputStream();

// to send data

OutputStream output = socket.getOutputStream();

Closing the Connection

The socket connection is closed explicitly once the message to the server is sent.

Example: Here, in the below program the Client keeps reading input from a user and sends it to the server until "Over" is typed.

```
// Demonstrating Client-side Programming
import java.io.*;
import java.net.*;

public class Client {

    // Initialize socket and input/output streams
    private Socket s = null;
    private DataInputStream in = null;
    private DataOutputStream out = null;

    // Constructor to put IP address and port
    public Client(String addr, int port)
    {
        // Establish a connection
        try {
            s = new Socket(addr, port);
            System.out.println("Connected");

            // Takes input from terminal
            in = new DataInputStream(System.in);

            // Sends output to the socket
            out = new DataOutputStream(s.getOutputStream());  }
        catch (UnknownHostException u) {
            System.out.println(u);
            return;
        }
        catch (IOException i) {
            System.out.println(i);
            return;
        }

        // String to read message from input
        String m = "";
```

```
        // Keep reading until "Over" is input
        while (!m.equals("Over")) {
            try {
                m = in.readLine();
                out.writeUTF(m);
            }
            catch (IOException i) {
                System.out.println(i);
            }
        }

        // Close the connection
        try {
            in.close();
            out.close();
            s.close();
        }
        catch (IOException i) {
            System.out.println(i);
        }
    }

    public static void main(String[] args) {
        Client c = new Client("127.0.0.1", 5000);
    }
}
```

Output:

```
java.net.ConnectException: Connection refused (Connection refused)
```

Explanation: In the above example, we have created a client program that establishes a socket connection to a server using an IP address and port, enabling data exchange. The client reads messages from the user and sends them to the server until the message “Over” is entered, after which the connection is closed.

Server-Side Programming

1. Establish a Socket Connection

To create a server application two sockets are needed.

- ServerSocket: This socket waits for incoming client requests. It listens for connections on a specific port.

- Socket: Once a connection is established, the server uses this socket to communicate with the client.

2. Communication

- Once the connection is established, you can send and receive data through the socket using streams.
- The getOutputStream() method is used to send data to the client.

3. Close the Connection

Once communication is finished, it's important to close the socket and the input/output streams to free up resources.

Example: The below Java program demonstrate the server-side programming

```
// Demonstrating Server-side Programming
import java.net.*;
import java.io.*;

public class Server {

    // Initialize socket and input stream
    private Socket s = null;
    private ServerSocket ss = null;
    private DataInputStream in = null;

    // Constructor with port
    public Server(int port) {

        // Starts server and waits for a connection
        try
        {
            ss = new ServerSocket(port);
            System.out.println("Server started");

            System.out.println("Waiting for a client ...");

            s = ss.accept();
            System.out.println("Client accepted");

            // Takes input from the client socket
            in = new DataInputStream(
                new BufferedInputStream(s.getInputStream()));

            String m = "";

            // Reads message from client until "Over" is sent
            while (!m.equals("Over"))
```

```
        {
            try
            {
                m = in.readUTF();
                System.out.println(m);

            }
            catch(IOException i)
            {
                System.out.println(i);
            }
        }
        System.out.println("Closing connection");

        // Close connection
        s.close();
        in.close();
    }
    catch(IOException i)
    {
        System.out.println(i);
    }
  }

  public static void main(String args[])
  {
    Server s = new Server(5000);
  }
}
```

Explanation: In the above example, we have implemented a server that listens on a specific port, accepts a client connection, and reads messages sent by the client. The server displays the messages until "Over" is received, after which it closes the connection and terminates.

Important Points:

- Server application makes a ServerSocket on a specific port which is 5000. This starts our Server listening for client requests coming in for port 5000.
- Then Server makes a new Socket to communicate with the client.

socket = server.accept()

- The accept() method blocks(just sits there) until a client connects to the server.
- Then we take input from the socket using getInputStream() method. Our Server keeps receiving messages until the Client sends "Over".

- After we're done we close the connection by closing the socket and the input stream.
- To run the Client and Server application on your machine, compile both of them. Then first run the server application and then run the Client application.

Run the Application

Open two windows one for Server and another for Client.

1. Run the Server

First run the Server application as:

$ java Server

Output:

Server started
Waiting for a client ...

2. Run the Client

Then run the Client application on another terminal as

$ java Client

Output:

Connected

3. Exchange Messages

- Type messages in the Client window.
- Messages will appear in the Server window.
- Type "Over" to close the connection.

Here is a sample interaction,

Client:

Hello

I made my first socket connection

Over

Server:

Hello

I made my first socket connection

Over

Closing connection

Notice that sending “Over” closes the connection between the Client and the Server just like said before.

Note : If you’re using Eclipse or likes of such:

1. Compile both of them on two different terminals or tabs
2. Run the Server program first
3. Then run the Client program
4. Type messages in the Client Window which will be received and shown by the Server Window simultaneously.
5. Type Over to end.

Java Networking Classes in Socket Programming

Java provides a set of classes in the java.net package that are essential for building network-based applications using sockets. These classes support both TCP and UDP communication.

---

1. Socket Class

- **Purpose:** Used by the **client** to connect to a server.
- **Protocol:** TCP (connection-oriented)
- **Key Methods:**
  - getInputStream()
  - getOutputStream()
  - close()

Example:

```
Socket socket = new Socket("localhost", 5000);

OutputStream out = socket.getOutputStream();

out.write("Hello Server".getBytes());

socket.close();
```

---

2. ServerSocket Class

- **Purpose:** Used by the **server** to listen for incoming TCP connections.
- **Protocol:** TCP
- **Key Methods:**
  - accept() – Waits for a client to connect
  - close()

Example:

```
ServerSocket serverSocket = new ServerSocket(5000);
Socket clientSocket = serverSocket.accept(); // Waits for client
InputStream in = clientSocket.getInputStream();
byte[] buffer = new byte[1024];
in.read(buffer);
System.out.println("Received: " + new String(buffer));
serverSocket.close();
```

---

3. DatagramSocket Class

- **Purpose:** Used for sending and receiving datagrams (UDP).
- **Protocol:** UDP (connectionless)
- **Key Methods:**
  - send(DatagramPacket)
  - receive(DatagramPacket)
  - close()

Example:

```
DatagramSocket socket = new DatagramSocket();
InetAddress address = InetAddress.getByName("localhost");
String msg = "Hello UDP";
DatagramPacket packet = new DatagramPacket(msg.getBytes(), msg.length(), address, 6000);
socket.send(packet);
socket.close();
```

---

4. DatagramPacket Class

- **Purpose:** Represents the data packet for UDP.
- **Used With:** DatagramSocket
- **Key Constructor:**
  - DatagramPacket(byte[] buf, int length, InetAddress address, int port) for sending.
  - DatagramPacket(byte[] buf, int length) for receiving.

Example:

```
byte[] buffer = new byte[1024];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
socket.receive(packet);
String received = new String(packet.getData(), 0, packet.getLength());
```

---

5. InetAddress Class

- **Purpose:** Represents an IP address (IPv4 or IPv6).
- **Usage:** To get or represent the address of the host.

Example:

```
InetAddress address = InetAddress.getByName("localhost");
System.out.println("IP Address: " + address.getHostAddress());
```

Summary Table:

Class	Use	Protocol	Role
Socket	Client connection	TCP	Client
ServerSocket	Wait for client connection	TCP	Server
DatagramSocket	Send/receive packets	UDP	Client/Server
DatagramPacket	Represents UDP data	UDP	Data Carrier
InetAddress	Represents IP address	TCP/UDP	Addressing

Multithreaded Server in Java

A multithreaded server handles multiple clients simultaneously. Each time a client connects, the server creates a new thread to handle that client, allowing concurrent communication.

Why Use It?

Without multithreading, the server can handle only one client at a time — other clients must wait.

Example:

```
// Server.java
import java.io.*;
import java.net.*;
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(5000);
        System.out.println("Server started...");

        while (true) {
            Socket socket = serverSocket.accept();
            new ClientHandler(socket).start(); // Handle client in new thread
        }
    }
}

class ClientHandler extends Thread {
    private Socket socket;

    ClientHandler(Socket socket) {
        this.socket = socket;
    }
    public void run() {
```

```
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            out.println("Hello from server!");
            String msg = in.readLine();
            System.out.println("Client says: " + msg);
            socket.close();
        } catch (IOException e) {
            System.out.println("Client error: " + e.getMessage());
        }
    }
}
```

---

2. Socket Exception and Error Handling

Network programs are prone to various runtime exceptions. Java provides robust handling through try-catch blocks. Proper error handling ensures your program doesn't crash unexpectedly.

Common Exceptions:

Exception	Cause
IOException	General I/O error (broken stream, etc.)
SocketException	Error during socket operation (e.g., closed socket)
UnknownHostException	Invalid host/IP address
ConnectException	Failed to connect to server
BindException	Port already in use

Example:

```
try {
    Socket socket = new Socket("localhost", 5000);
```

```
    // Communication logic here
    socket.close();
} catch (UnknownHostException e) {
    System.out.println("Host not found: " + e.getMessage());
} catch (ConnectException e) {
    System.out.println("Connection failed: " + e.getMessage());
} catch (SocketException e) {
    System.out.println("Socket error: " + e.getMessage());
} catch (IOException e) {
    System.out.println("I/O error: " + e.getMessage());
}
```

Real-World Applications

1.Chat Applications

Chat applications (like WhatsApp Web or Slack) use sockets to enable **real-time text communication** between users. Each client connects to a central server using a TCP socket. The server receives and forwards messages to the intended recipient(s).

Use of Sockets:

- TCP ensures reliable message delivery.
- Each user has a dedicated socket connection to the server.
- Often implemented using multithreaded servers for handling multiple users simultaneously.

---

2. Multiplayer Games

In online multiplayer games (e.g., PUBG, Minecraft), socket programming is used to **exchange game state information** like player movement, scores, and events in real time.

Use of Sockets:

- UDP is often preferred for fast, low-latency communication.
- TCP may be used for login or score synchronization where reliability matters.
- Sockets allow continuous data flow between clients and the game server.

---

3. File Transfer Tools

Tools like FTP clients or custom file transfer systems use sockets to send and receive files across a network. The sender reads a file and sends its byte stream, while the receiver writes it to disk.

Use of Sockets:

- TCP is used for reliable and ordered transmission of file data.
- Large files are often split into chunks and sent through the socket stream.
- Java's InputStream and OutputStream are used to handle file data over sockets.

---

4. Web Servers and Clients

Web browsers and servers (like Apache or custom HTTP servers in Java) communicate using sockets based on the **HTTP protocol** over TCP.

Use of Sockets:

- Browsers open TCP sockets to send HTTP requests.
- Servers listen for connections via ServerSocket and respond with HTML content.
- Low-level socket programming is used in custom server or client development.

Advantages of Socket Programming in Java

1. Real-time Communication: Enables real-time communication between client and server.

2. Networked Applications: Allows creation of networked applications, such as chat apps, online gaming, and more.

3. Flexibility: Supports both TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

4. Platform Independence: Java's socket programming is platform-independent, making it versatile.

Disadvantages of Socket Programming in Java

1. Complexity: Socket programming can be complex, requiring understanding of network protocols.

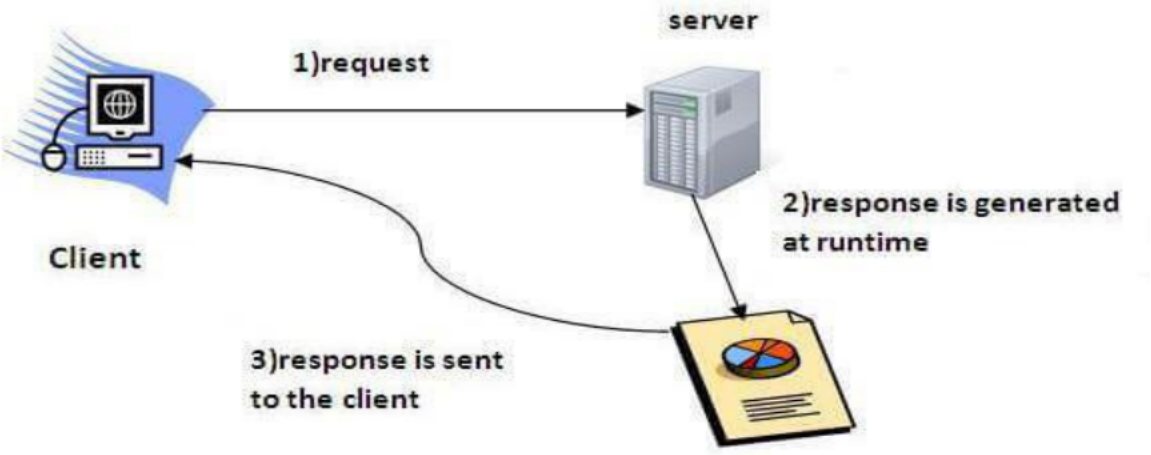
2. Error Handling: Requires robust error handling mechanisms.

3. Security: Sockets can be vulnerable to security threats if not properly secured.

4. Performance: Can be resource-intensive, impacting performance.

SERVLET

Servlet is simple java program that runs on server and capable of handling request and generating dynamic response.



Servlet Lifecycle:

The lifecycle of a servlet is controlled by the container (e.g., Tomcat), and it goes through the following stages:

1. Loading and Instantiation: The servlet class is loaded and an instance is created.

2. Initialization (init method): The init method is called once to initialize the servlet.

3. Request Handling (service method): The service method is called for each request to process it.

4. Destruction (destroy method): The destroy method is called once before the servlet is removed from service.

Steps to Create a Servlet:

1.Create a class that extends HttpServlet.

2.Override doGet() or doPost() methods.

3.Compile and deploy the servlet in a servlet container (e.g., Apache Tomcat).

4.Configure the servlet in web.xml or using @WebServlet annotation.

Setting up the environment

Requirement tools

- **JDK:** Java Development Kit
- **Apache Tomcat:** Web server and servlet container
- **Eclipse/IntelliJ IDEA:** Integrated development Environment(IDE)
- Configure tomcat with IDE.

Example:

```
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<h1>Hello, Servlet!</h1>");
    }
}
```

Uses of Servlets:

- **Building dynamic web applications**
- **Handling form submissions**
- **Creating REST APIs**
- **Managing sessions and cookies**

Advantages:

- **Platform independent**
- **Efficient and scalable**
- **Secure and robust**

JSP

Java Server Pages (JSP) is a server-side technology that creates dynamic web applications. It allows developers to embed Java code directly into HTML or XML pages, and it makes web development more efficient.

JSP is an advanced version of Servlets. It provides enhanced capabilities for building scalable and platform-independent web pages.

Key Features of JSP

- It is platform-independent; we can write once, run anywhere.
- It simplifies database interactions for dynamic content.
- It contains predefined objects like request, response, session, and application, reducing development time.
- It has built-in mechanisms for exception and error management.
- It supports custom tags and tag libraries.

How JSP Works

- A JSP file is an HTML page with Java code.
- When a client requests a JSP, the server compiles it into a Servlet.
- The Servlet handles the request, processes the logic, and returns an HTML response to the client.

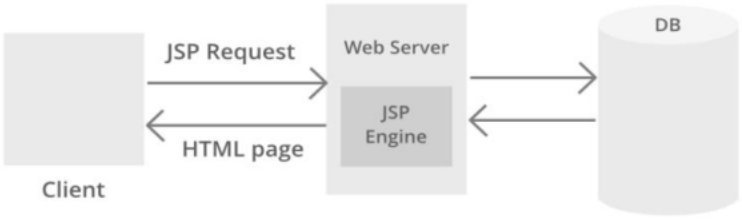
JSP Architecture

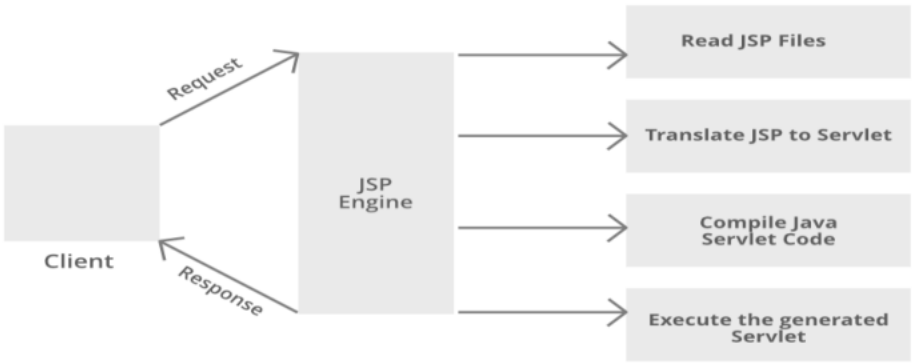
JSP Architecture gives a high-level view of the working of JSP. JSP architecture is a 3 tier architecture. It has a Client, Web Server, and Database. The client is the web browser or application on the user side. Web Server uses a JSP Engine i.e; a container that processes JSP. For example, Apache Tomcat has a built-in JSP Engine. JSP Engine intercepts the request for JSP and provides the runtime environment for the understanding and processing of JSP files. It reads, parses, build Java Servlet, Compiles and

Executes Java code, and returns the HTML page to the client. The webserver has access to the Database. The following diagram shows the architecture

JSP follows a three-layer architecture:

- **Client Layer**: The browser sends a request to the server.
- **Web Server Layer**: The server processes the request using a JSP engine.
- **Database/Backend Layer**: Interacts with the database and returns the response to the client.





Step-by-Step Flow

1. **Client Request (Browser)**
    - The user requests a .jsp page by entering a URL or submitting a form.
    - HTTP request is sent to the **web server**.
2. **Web Server / Servlet Container**

- The request reaches the **JSP engine** inside the Servlet container (e.g., Apache Tomcat).

3. **JSP Translation**

- If this is the first request or the JSP has changed:
  - The JSP file is translated into a **Servlet** (Java class).

4. **JSP Compilation**

- The translated Servlet is compiled into a .class file (bytecode).

5. **Servlet Execution**

- The Servlet is loaded into memory and initialized.
- The jspService() method is called to process the request.

6. **Business Logic (Optional)**

- The Servlet (from JSP) may interact with **JavaBeans**, **JDBC**, or **backend classes** to retrieve or update data.

7. **Response Generation**

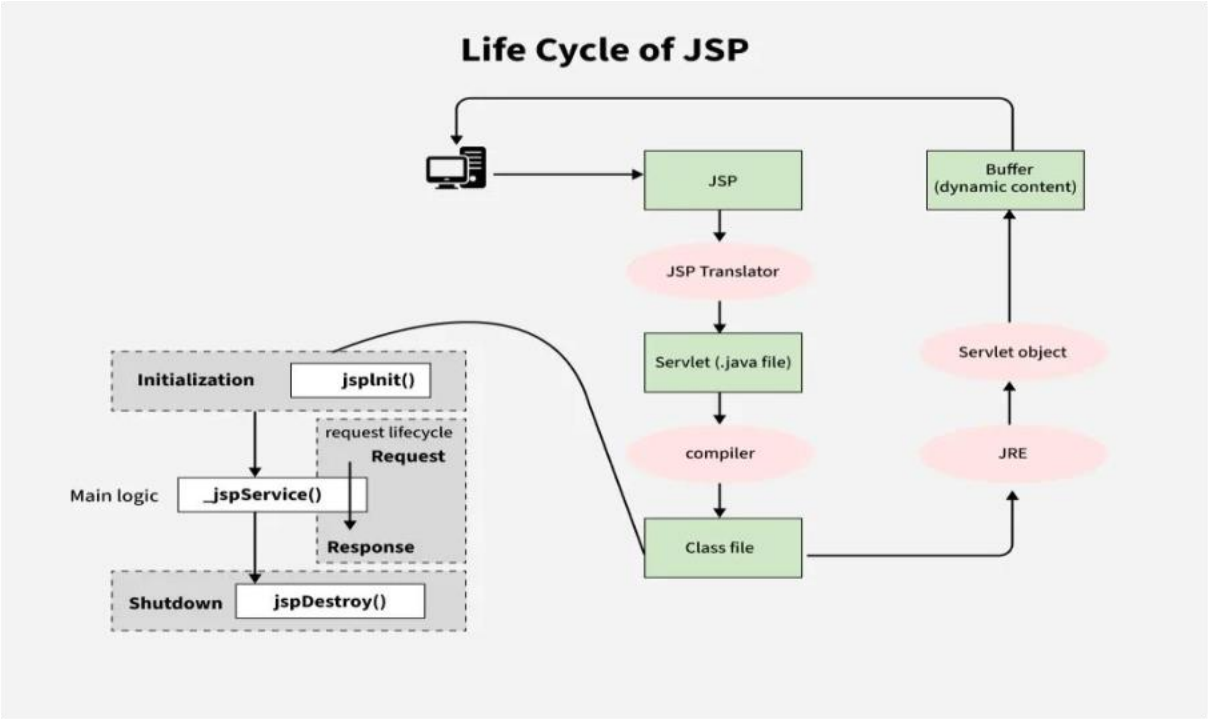
- The Servlet produces **HTML content** using data and sends it back to the browser.

8. **Client Response**

- The client receives a fully rendered HTML page.

Life Cycle of JSP

The life cycle of a JavaServer Page (JSP) consists of various phases that start from its creation, followed by its translation into a servlet, and finally managed by the servlet lifecycle. The JSP engine handles this process automatically.



Steps of JSP Life Cycle

1. Translation Phase

- The JSP page is translated into a **Servlet** (Java source file).
- This is done by the **JSP Engine** inside the servlet container (e.g., Apache Tomcat).
- Translation occurs **only once**, unless the JSP page is modified.

2. Compilation Phase

- The generated Servlet is **compiled into a .class file** (Java bytecode).
- This Servlet class can now handle client requests.

3. Initialization Phase

- The container **loads the compiled Servlet**.
- The **jspInit()** method is called once for initialization (like database connections, resource loading).

4. Request Handling (Execution) Phase

- For every client request, the container invokes the **_jspService(HttpServletRequest, HttpServletResponse)** method.

- This method contains the logic defined in the JSP page (translated into Java).
- It generates a **dynamic response** (usually HTML) and sends it to the client.

5. Destruction Phase

- When the container decides to remove the JSP/Servlet (e.g., server shutdown or redeployment), it calls the **jspDestroy()** method.
- Used to **release resources** (e.g., closing database connections).

```
public void jspDestroy() {

   // Clean up resources like closing database connections.
System.out.println("JSP Destroyed.");
}
```

This Lifecycle ensures that JSP pages are efficiently compiled, managed and cleaned up by the server container.

Flow Chart Description (Text)

```
Client Request
   ↓
Translate JSP to Servlet
   ↓
Compile Servlet (.java → .class)
   ↓
Load Servlet and call jspInit()
   ↓
Handle request using _jspService()
   ↓
Send Response to Client
   ↓
On shutdown or unload → jspDestroy()
```

Adding Dynamic Content with JSP

Here is an example to demonstrate how JSP can generate dynamic content:

hello.jsp:

```
<!DOCTYPE html>
<html>
```

```
<body>
   Hello! The time is now <%= new java.util.Date() %>
</body>
</html>
```

Example of a JSP Web Page

Example:

```
<!DOCTYPE html>
<html>
<head>
   <title>A Web Page</title>
</head>
<body>
   <% out.println("Hello there!"); %>
</body>
</html>
```

Why We Use JSP

1. Dynamic Content Generation

JSP allows embedding Java code into HTML, enabling the creation of **dynamic web pages** (e.g., user-specific content, data from databases, etc.).

2. Simplified Web Development

- Developers can write **HTML and Java code in one file**, making it easy to design pages.
- No need to write complex Java Servlets just to display content.

3. Automatic Compilation to Servlet

- JSPs are **converted into Servlets automatically** by the server (e.g., Tomcat).
- This combines the power of Servlets with simpler syntax.

5. Built-in Features

- JSP supports **session management**, **form handling**, and **request objects**.
- Provides **implicit objects** like request, response, session, application, etc.

7. Reusable Components

- JSP supports **tag libraries** (like JSTL), custom tags, and includes — making code modular and reusable.

The difference between Servlet and JSP is as follows:

Servlet	JSP
Servlet is a java code.	JSP is a HTML-based compilation code.
Writing code for servlet is harder than JSP as it is HTML in java.	JSP is easy to code as it is java in HTML.
Servlet plays a controller role in the ,MVC approach.	JSP is the view in the MVC approach for showing output.
Servlet is faster than JSP.	SP is slower than Servlet because the first step in the JSP lifecycle is the translation of JSP to java code and then compile.
Servlet can accept all protocol requests.	JSP only accepts HTTP requests.
In Servlet, we can override the service() method.	In JSP, we cannot override its service() method.
It can handle extensive data processing.	It cannot handle extensive data processing very efficiently.
The facility of writing custom tags is not present.	The facility of writing custom tags is present.
It does not have inbuilt implicit objects.	In JSP there are inbuilt implicit objects.
There is no method for running JavaScript on the client side in Servlet.	While running the JavaScript at the client side in JSP, client-side validation is used.
Packages are to be imported on the top of the program.	Packages can be imported into the JSP program (i.e, bottom , middleclient-side, or top)

ABSTRACT WINDOW TOOLKIT

- AWT is the set of java classes that allows us to create GUI (Graphical User Interfaces) component and manipulate them.
- GUI Components are used to take the input from user in a user friendly manner.

Container

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains.

Panel

The **Panel** class is a concrete subclass of **Container**. It doesn't add any new methods; it simply implements **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( )**, **setSize( )**, or **setBounds( )** methods defined by **Component**.

Window

The **Window** class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. If you create a **Frame** object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. (An applet that could masquerade as a host-based application could be used to obtain passwords and other sensitive information without the user's knowledge.) When a **Frame** window is created by a program rather than an applet, a normal window is created.

Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. **Canvas** encapsulates a blank window upon which you can draw. You will see an example of **Canvas** later in this book.

Here are two of **Frame**'s constructors:
Frame( )
Frame(String title)
The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by title.

There are several methods you will use when working with **Frame** windows.
The **setSize( )** method is used to set the dimensions of the window. Its signature is shown here: void setSize(int newWidth, int newHeight)

After a frame window has been created, it will not be visible until you call **setVisible( )**. Its signature is shown here: void setVisible(boolean visibleFlag)
The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

You can change the title in a frame window using **setTitle( )**, which has this general form: void setTitle(String newTitle)
Here, newTitle is the new title for the window.

Control Fundamentals

The AWT supports the following types of controls:

1. Label

A label is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. Label defines the following constructors:

Label( )	creates a blank label
Label(String str)	creates a label that contains the string specified by str
Label(String str, int how)	creates a label that contains the string specified by str using the alignment specified by how. The value of how must be one of these three constants: Label.LEFT,Label.RIGHT, or

	Label.CENTER.

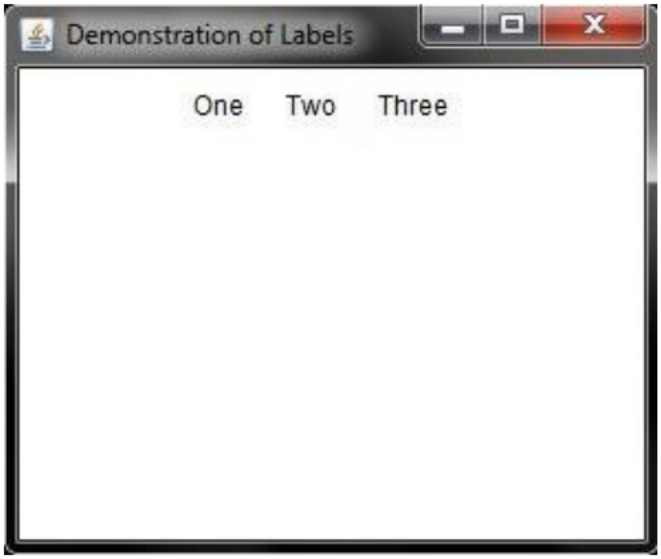
We can set or change the text in a label by using the **setText( )** method.We can obtain the current label by calling **getText( )**. These methods are:
void setText(String str)
String getText( )

Program Demonstration of Label.

Solution:

```
import java.awt.*;
class LabelDemo extends Frame
{
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");
        public LabelDemo(String s)
        {
        super(s);
        setLayout(new FlowLayout());
        add(one);
        add(two);s
        add(three);
        }

        public static void main(String args[])
        {
        LabelDemo l=new LabelDemo("Demonstration of Labels");
        l.setSize(300, 250);
        l.setVisible(true);
        }

}
```

Output:



2. Button

A button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type Button. Button defines these two constructors:

Button( )	creates an empty button
Button(String str)	creates a button that contains str as a label

After a button has been created, we can set its label by calling **setLabel( )**. We can retrieve its label by calling **getLabel( )**. These methods are as follows:

void setLabel(String str)

String getLabel( )

Here, str becomes the new label for the button.

Program

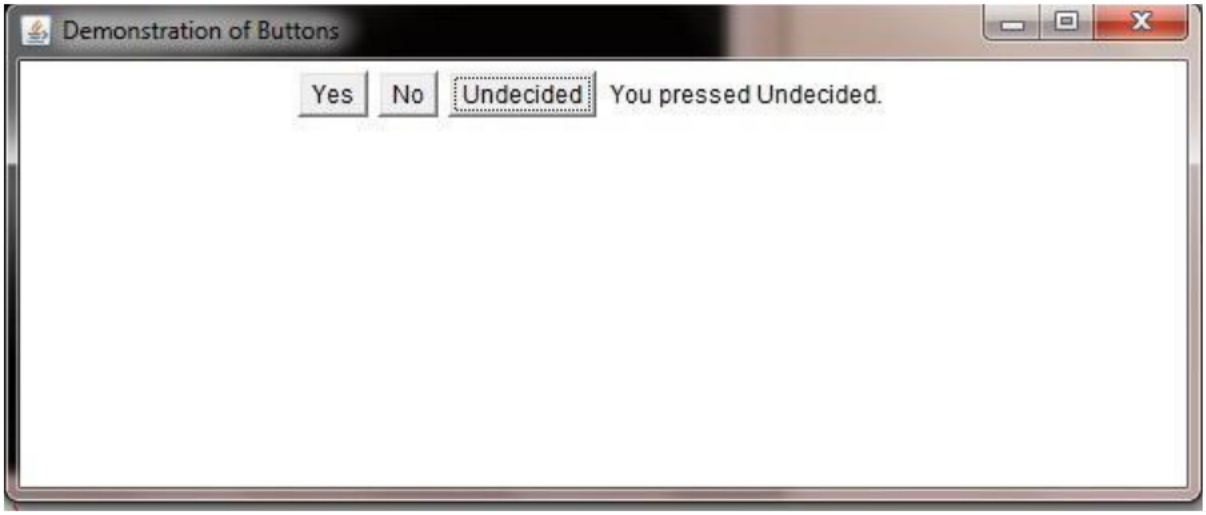
Demonstration of Button.

Solution:

```
import java.awt.*;
import java.awt.event.*;
class ButtonDemo extends Frame implements ActionListener {

        String msg = " ";
        Button yes= new Button("Yes");
        Button no= new Button("No");
        Button maybe= new Button("Undecided");
        Label l=new Label(msg);
        public ButtonDemo(String s)
```

```
{
super(s);
setLayout(new FlowLayout());
add(yes);
add(no);
add(maybe);add(l);
yes.addActionListener(this);
no.addActionListener(this);
maybe.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
{
String str = ae.getActionCommand();
if(str.equals("Yes"))
{
msg = "You pressed Yes.";
}
else if(str.equals("No"))
{
msg = "You pressed No.";
}
else
{
msg = "You pressed Undecided.";
}
l.setText(msg);
}
public static void main(String args[])
{
ButtonDemo b=new ButtonDemo("Demonstration of Buttons");
b.setSize(600, 250);
b.setVisible(true);
}
}
```

Output:



3. Checkbox

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. Checkbox supports these constructors:

Checkbox( ) blank	createsacheckbox whoselabelis initially
Checkbox(String str)	createsacheckbox whoselabelis specified by str
Checkbox(String str, boolean on)	set the initial state of the check box. If on is **true,** the check box is initially checked; otherwise, it is cleared.
Checkbox(String str, boolean on, CheckboxGroup cbGroup)	create a check box whose label is specified by str and whose group is specified by cbGroup. If this check box is not part of a group, then cbGroup must be **null**.
Checkbox(String str, CheckboxGroup cbGroup, boolean on)	

To retrieve the current state of a check box, call getState( ). To set its state, call setState( ). We can obtain the current label associated with a check box by calling getLabel( ). To set the label, call setLabel( ). These methods are as follows:

boolean getState( )

void setState(boolean on)
String getLabel( )

void setLabel(String str)

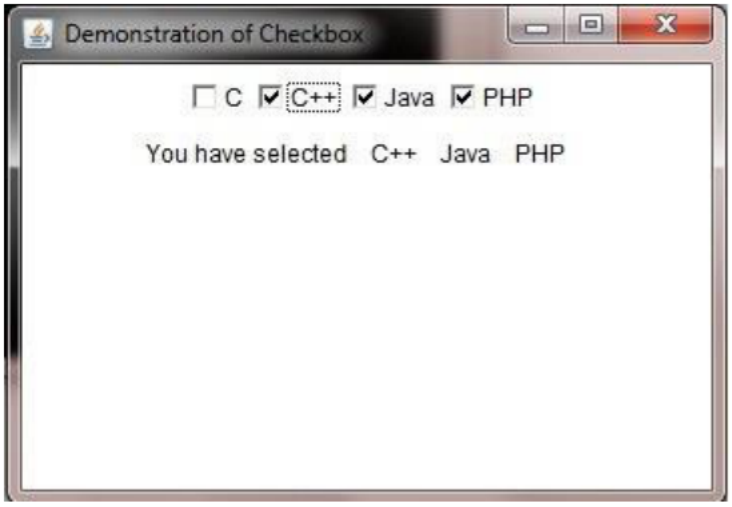
Program Demonstration of Checkbox.

Solution:

```
import java.awt.*;
import java.awt.event.*;
class CheckboxDemo extends Frame implements ItemListener {

        String msg = " ";
        Checkbox c1 = new Checkbox("C", null, true);
        Checkbox c2 = new Checkbox("C++");
        Checkbox c3 = new Checkbox("Java");
        Checkbox c4 = new Checkbox("PHP");
        Label l=new Label(msg);
        public CheckboxDemo(String s)
        {

        super(s);
        setLayout(new FlowLayout());
        add(c1);
        add(c2);
        add(c3);
        add(c4);
        add(l);
        c1.addItemListener(this);
        c2.addItemListener(this);
        c3.addItemListener(this);
        c4.addItemListener(this);
        }

        public void itemStateChanged(ItemEvent ie)
        {

        msg="You have selected";
        if(c1.getState())
        msg=msg+" C";
        if(c2.getState())
        msg=msg+" C++";
        if(c3.getState())
        msg=msg+" Java";
        if(c4.getState())
        msg=msg+" PHP";
        l.setText(msg);
        }
```

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

        CheckboxDemo b=new CheckboxDemo ("Demonstration of Checkbox");
        b.setSize(350, 250);
        b.setVisible(true);
        }

}
```

Output:



4. Radio Buttons/ Checkbox Group

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called radio buttons, because they act like the station selector on a car radio—only one station can be selected at any one time. We can determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**. We can set a check box by calling **setSelectedCheckbox( )**. These methods are as follows:

Checkbox getSelectedCheckbox( )

void setSelectedCheckbox(Checkbox which)

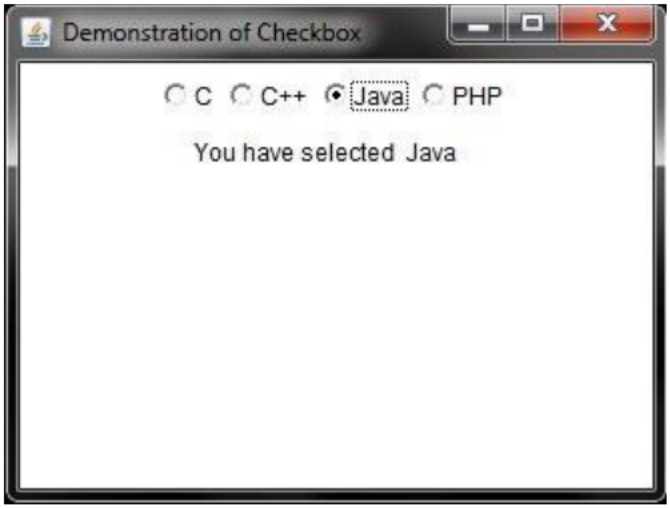
[redacted] Demonstration of Radio Buttons.

Solution:

```
import java.awt.*;
import java.awt.event.*;
class RadioDemo extends Frame implements ItemListener {
```

```
        String msg = " ";

        CheckboxGroup cbg=new CheckboxGroup();
        Checkbox c1 = new Checkbox("C", cbg, true);
        Checkbox c2 = new Checkbox("C++",cbg,false);
        Checkbox c3 = new Checkbox("Java",cbg,false);
        Checkbox c4 = new Checkbox("PHP",cbg,false);
        Label l=new Label(msg);
        public RadioDemo(String s)
        {

        super(s);
        setLayout(new FlowLayout());
        add(c1);
        add(c2);
        add(c3);
        add(c4);
        add(l);
        c1.addItemListener(this);
        c2.addItemListener(this);
        c3.addItemListener(this);
        c4.addItemListener(this);
        }

        public void itemStateChanged(ItemEvent ie)
        {

        msg="You have selected "+cbg.getSelectedCheckbox().getLabel();
        l.setText(msg);
        }

        public static void main(String args[])
        {

        RadioDemo b=new RadioDemo ("Demonstration of Checkbox");
        b.setSize(350, 250);
        b.setVisible(true);
        }

}
```

Output:



5. Choice

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object. Choice only defines the default constructor, which creates an empty list.

To add a selection to the list, call add( ). It has this general form:

void add(String name)

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur.

Todeterminewhichitemiscurrentlyselected,wemaycalleithergetSelectedItem()or getSelectedIndex( ). These methods are shown here:

String getSelectedItem( )

int getSelectedIndex( )

The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index o. By default, the first item added to the list is selected. To obtain the number of items in the list, call getItemCount( ). We can set the currently selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

int getItemCount( )

void select(int index)

void select(String name)

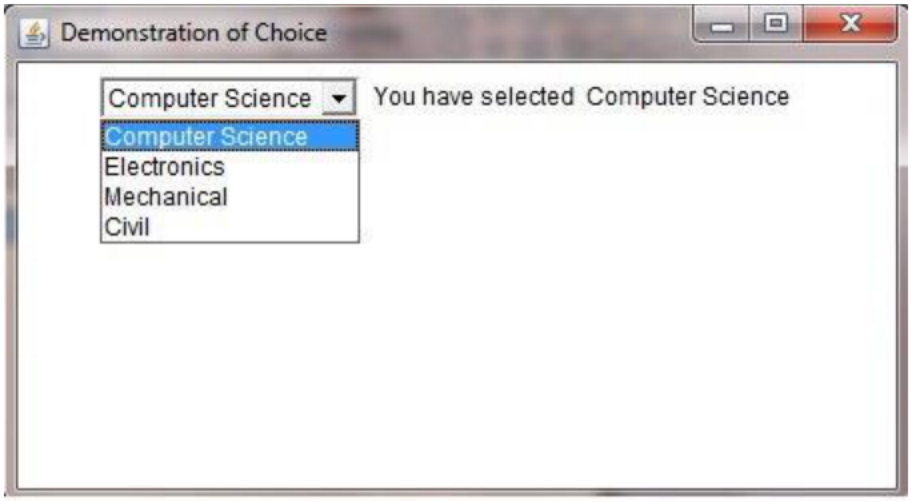
Program Demonstration of Choice.

Solution:

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

```
class ChoiceDemo extends Frame implements ItemListener {

        String msg = " ";
        Choice dept=new Choice();
        Label l=new Label(msg);

        public ChoiceDemo(String s)
        {

        super(s);
        setLayout(new FlowLayout());
        dept.add("Computer Science");
        dept.add("Electronics");
        dept.add("Mechanical");
        dept.add("Civil");
        add(dept);
        add(l);
        dept.addItemListener(this);
        }


        public void itemStateChanged(ItemEvent ie)
        {

        msg="You have selected "+dept.getSelectedItem();
        l.setText(msg);
        }


        public static void main(String args[])
        {

        ChoiceDemo b=new ChoiceDemo("Demonstration of Choice");
        b.setSize(450, 250);
        b.setVisible(true);
        }

}
```

Output:



6. List

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

List( )	creates a **List** control that allows only one item to be selected at any one time
List(int numRows)	the value of numRows specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed)
List(int numRows, boolean multipleSelect)	if multipleSelect is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

To add a selection to the list, call **add( )**. It has the following two forms: void add(String name)
void add(String name, int index)

Here, name is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by index. Indexing begins at zero. You can specify –1 to add the item to the end of the list. For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:
String getSelectedItem( )
int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, **null** is returned. **getSelectedIndex( )** returns the index

of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

For lists that allow multiple selection, you must use either **getSelectedItems( )** or **getSelectedIndexes ( )**, shown here, to determine the current selections:

String[ ] getSelectedItems( )
int[ ] getSelectedIndexes( )

getSelectedItems()returnsanarraycontainingthenamesofthecurrentlyselected items.

getSelectedIndexes( ) returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item by using the **select( )** method with a zero-based integer index. These methods are shown here: int getItemCount( )
void select(int index)
Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:
String getItem(int index)
Here, index specifies the index of the desired item.

Demonstration of List box.

[redacted]

```
import java.awt.*;
import java.awt.event.*;
class ListDemo extends Frame implements ItemListener {

        String msg = " ";
        List dept=new List(4, true);
        Label l=new Label(msg);

        public ListDemo(String s)
        {

        super(s);
        setLayout(new FlowLayout());
        dept.add("Computer Science");
        dept.add("Electronics");
        dept.add("Mechanical");
        dept.add("Civil");
        add(dept);
        add(l);
        dept.addItemListener(this);
        }
```

```
public void itemStateChanged(ItemEvent ie)
{
int idx[];
msg="You have selected ";
idx = dept.getSelectedIndexes();
for(int i=0; i<idx.length; i++)
msg += dept.getItem(idx[i]) + " ";
l.setText(msg);}

public static void main(String args[])
{
ListDemo b=new ListDemo("Demonstration of List");
b.setSize(450, 250);
b.setVisible(true);
}
}
```

7. TextField

The **TextField** class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. **TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:

TextField( )	creates a default text field
TextField(int numChars)	creates a text field that is numChars characters wide
TextField(String str)	initializes the text field with the string contained in str
TextField(String str, int numChars)	initializes a text field and sets its width.

TextField (and its superclass TextComponent) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call getText( ). To set the text, call setText( ). These methods are as follows:

String getText( )
void setText(String str)s
Here, str is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using select( ). Your program can obtain the currently selected text by calling getSelectedText( ). These methods are shown here:

String getSelectedText( )

void select(int startIndex, int endIndex)

getSelectedText( ) returns the selected text. The select( ) method selects the characters beginning at startIndex and ending at endIndex–1. You can control whether the contents of a text field may be modified by the user by calling setEditable( ). You can determine editability by calling isEditable( ). These methods are shown here:

boolean isEditable( )

void setEditable(boolean canEdit)

isEditable( ) returns true if the text may be changed and false if not. In setEditable( ), if canEdit is true, the text may be changed. If it is false, the text cannot be altered. There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the echoCharIsSet( ) method. You can retrieve the echo character by calling the getEchoChar( ) method.

These methods are as follows:

void setEchoChar(char ch)

boolean echoCharIsSet( )

char getEchoChar( )

Here, ch specifies the character to be echoed.

8. TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea. Following are the constructors for TextArea: TextArea( )

TextArea(int numLines, int numChars)

TextArea(String str)

TextArea(String str, int numLines, int numChars)

TextArea(String str, int numLines)

Layout Managers

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout ( )** method. If no call to **setLayout ( )** is made, then the default layout manager is used. Whenever a

container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The **setLayout( )** method has the following general form:

void setLayout(LayoutManager layoutObj)

Java has several predefined **LayoutManager** classes, several of which are: FlowLayout, BorderLayout, GridLayout.

FlowLayout

FlowLayout is the default layout manager. This is the layout manager that the preceding examples have used. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for **FlowLayout**:

FlowLayout( )	creates the default layout, which centers components and leaves five pixels of space between each component
FlowLayout(int how)	specify how each line is aligned. Valid values for how are as follows: FlowLayout.LEFT FlowLayout.CENTER FlowLayout.RIGHTs
FlowLayout(int how, int horz, int vert)	specifythehorizontalandverticalspaceleft between components in horz and vert, respectively.

BorderLayout

The BorderLayout class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by BorderLayout:

BorderLayout( )	creates a default border layout
BorderLayout(int horz, int vert)	specify the horizontal and verticalspace left between components in horz and vert, respectively

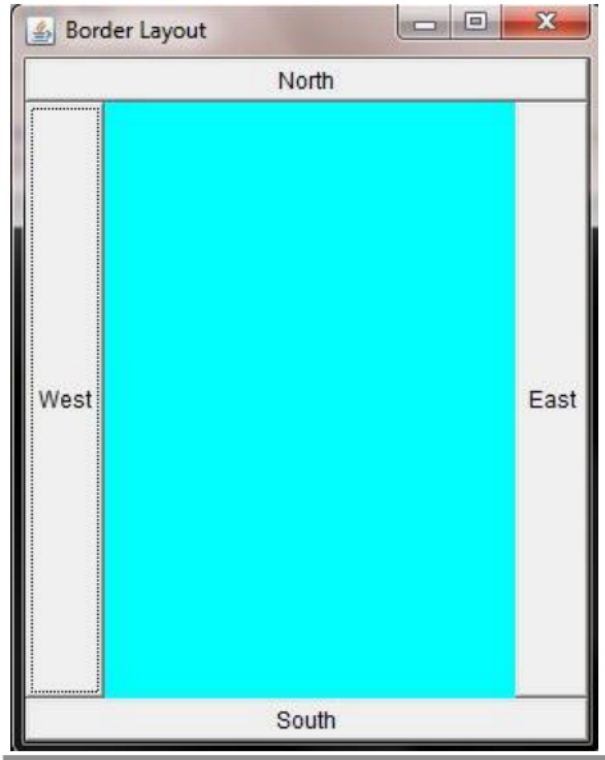
BorderLayout defines the following constants that specify the regions:
BorderLayout.CENTER
BorderLayout.SOUTH
BorderLayout.EAST
BorderLayout.WEST
BorderLayout.NORTH
When adding components, you will use these constants with the following form of add( ), which is defined by Container:
void add(Component compObj, Object region);
Here, compObj is the component to be added, and region specifies where the component will be added.

Demonstration of Border Layout.

Solution:

```
import java.awt.*;
class Border extends Frame
{
        Button b1=new Button("East");
        Button b2=new Button("West");
        Button b3=new Button("North");
        Button b4=new Button("South");
        public Border(String s1)
        {
        super(s1);
        setBackground(Color.cyan);
        setLayout(new BorderLayout());
        add(b1, BorderLayout.EAST);
        add(b2, BorderLayout.WEST);
        add(b3, BorderLayout.NORTH);
        add(b4, BorderLayout.SOUTH);
        }
        public static void main(String args[])
        {
        Border c=new Border("Border Layout");
        c.setSize(300,400);
        c.show();
        }
}
```

Output:



GridLayout

GridLayout lays out components in a two-dimensional grid. When you instantiate a GridLayout, you define the number of rows and columns. The constructors supported by GridLayout are shown here:

GridLayout( )	creates a single-column grid layout
GridLayout(int numRows, int numColumns) and columns	creates a grid layout with the specified number of rows
GridLayout(int numRows, int numColumns, int horz, int vert)	specifythehorizontalandverticalspaceleft between components in horz and vert, respectively.

Program

Demonstartion of GridLayout.

Solution:

```
import java.awt.*;
class Cal extends Frame
{
        String s=new
        String("123+456-789*.=0/");
        int i;
        Button b[]=new
        Button[16]; public
        Cal(String s1)
```

SWING IN JAVA

Java Swing Tutorial

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is used to create window-based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in Java.

Java Swing, a graphical user interface (GUI) toolkit, has been a cornerstone of Java development for decades. Since its inception, Swing has provided Java developers with a robust framework for creating interactive, platform-independent applications. In this article, we'll delve into the fundamentals of Java Swing, explore its key features, and discuss its relevance in modern application development.

In this section, we will discuss the fundamentals of Java Swing, explore its key features, and discuss its relevance in modern application development.

Java Swing

Java Swing was introduced as part of the Java Foundation Classes (JFC) in the late 1990s, aiming to address the limitations of the earlier Abstract Window Toolkit (AWT). Unlike AWT, that relies on the native platform's components for rendering, Swing is entirely written in Java, offering a consistent look and feel across different operating systems.

Swing provides a comprehensive set of components for building GUIs, including buttons, text fields, panels, and more. These components are highly customizable, allowing developers to create visually appealing and user-friendly interfaces.

Key Features:-

- Simple Java SwinFeatures Of Swing Class
- Pluggable look and feel.
- Uses MVC architecture.
- Lightweight Components
- Platform Independent
- Advanced features such as JTable, JTabbedPane, JScollPane, etc.
- Java is a platform-independent language and runs on any client machine, the GUI look and feel, owned and delivered by a platform-specific O/S, simply does not affect an application's GUI constructed using Swing components.

- Lightweight Components: Starting with the JDK 1.1, its AWT-supported lightweight component development. For a component to qualify as lightweight, it must not depend on any non-Java [O/s based) system classes. Swing components have their own view supported by Java's look and feel classes.
- Pluggable Look and Feel: This feature enable the user to switch the look and feel of Swing components without restarting an application. The Swing library supports components' look and feels that remain the same across all platforms wherever the program runs. The Swing library provides an API that gives real flexibility in determining the look and feel of the GUI of an application
- Highly customizable – Swing controls can be customized in a very easy way as visual appearance is independent of internal representation.
- Rich controls– Swing provides a rich set of advanced controls like Tree TabbedPane, slider, colorpicker, and table controls**.**

PROGRAM: Simple Java Swing Program:-

SimpleSwingApp.java

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

public class SimpleSwingApp {
   public static void main(String[] args) {
      // Create a new JFrame
      JFrame frame = new JFrame("Simple Swing App");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a label
      JLabel label = new JLabel("Hello, World!");
      label.setFont(new Font("Arial", Font.BOLD, 24));

      // Add the label to the frame
      frame.getContentPane().add(label, BorderLayout.CENTER);

      // Set the frame size
      frame.setSize(400, 300);

         // Center the frame on the screen
         frame.setLocationRelativeTo(null);

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

```
    }
}
```

Output

A window with the title "Simple Swing App" and a label that says "Hello, World!" in bold Arial font.

How it Works:-

1. We create a JFrame (window) and set its default close operation.

2. We create a JLabel and set its font.

3. We add the label to the frame's content pane.

4. We set the frame's size and location.

5. Finally, we make the frame visible.

This code demonstrates the basics of creating a GUI application with Java Swing.

PROGRAM: first code java swing layout

```
// Java program to illustrate the CardLayout Class
import java.awt.*;
import java.awt.event.*;
import javax.swing.JFrame;
import javax.swing.*;

// class extends JFrame and implements actionlistener
public class Cardlayout extends JFrame implements ActionListener {

    // Declaration of objects of CardLayout class.
    CardLayout card;

    // Declaration of objects of JButton class.
    JButton b1, b2, b3;

    // Declaration of objects
    // of Container class.
    Container c;

    Cardlayout()
    {
```

```
    // to get the content
    c = getContentPane();

    // Initialization of object "card"
    // of CardLayout class with 40
    // horizontal space and 30 vertical space .
    card = new CardLayout(40, 30);

    // set the layout
    c.setLayout(card);

    // Initialization of object "b1" of JButton class.
    b1 = new JButton("GEEKS");

    // Initialization of object "b2" of JButton class.
    b2 = new JButton("FOR");

    // Initialization of object "b3" of JButton class.
    b3 = new JButton("GEEKS");

    // this Keyword refers to current object.
    // Adding Jbutton "b1" on JFrame using ActionListener.
    b1.addActionListener(this);

    // Adding Jbutton "b2" on JFrame using ActionListener.
    b2.addActionListener(this);

    // Adding Jbutton "b3" on JFrame using ActionListener.
    b3.addActionListener(this);

    // Adding the JButton "b1"
    c.add("a", b1);

    // Adding the JButton "b2"
    c.add("b", b2);

    // Adding the JButton "b1"
    c.add("c", b3);
}

public void actionPerformed(ActionEvent e)
{

    // call the next card
```

```
        card.next(c);
    }

    // Main Method
    public static void main(String[] args)
    {

        // Creating Object of CardLayout class.
        Cardlayout cl = new Cardlayout();

        // Function to set size of JFrame.
        cl.setSize(400, 400);

        // Function to set visibility of JFrame.
        cl.setVisible(true);

        // Function to set default operation of JFrame.
    cl.setDefaultCloseOperation(EXIT_ON_CLOSE);          }
}
```

Applications of Java Swing

While newer GUI frameworks such as JavaFX and web technologies like React and Angular have gained popularity in recent years, Java Swing remains a viable choice for certain types of applications:

- **Desktop Applications:** Swing is well-suited for developing desktop applications that require a rich and responsive user interface. Applications like IDEs (Integrated Development Environments), productivity tools, and multimedia players can benefit from Swing's extensive feature set.
- **Legacy Systems:** Many enterprise applications built with Swing continue to be in use today. While these applications may not leverage the latest technologies, Swing provides a stable and reliable platform for maintaining and extending existing systems.
- **Educational Purposes:** Swing is often used as a teaching tool in computer science courses to introduce students to GUI programming concepts. Its simplicity and versatility make it an ideal choice for beginners learning Java programming.

Difference between Java Swing and Java AWT

There are certain points from which Java Swing is different than Java AWT as mentioned below:

Java AWT	Java Swing
AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
AWT components are **heavyweight**.	Swing components are **lightweight**.
AWT **does not support pluggable look and feel**.	Swing **supports pluggable look and feel**.
AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
AWT **does not follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing follows **MVC**.

JFC:-

JFC (Java Foundation Classes) is a **set of APIs** provided by Java for building **rich graphical user interfaces (GUIs)**. It is a part of the Java platform and extends the Abstract Window Toolkit (AWT) to provide more powerful and flexible UI components.

Key Components of JFC:

1. **Swing**
   - A set of lightweight GUI components like JButton, JLabel, JTable, etc.
   - Platform-independent and supports pluggable look-and-feel.
2. **AWT (Abstract Window Toolkit)**
   - Provides basic GUI components using native system resources (heavyweight).
   - Forms the base for Swing components.
3. **Accessibility API**
   - Helps make GUI components accessible to users with disabilities.

4.**Java 2D API**

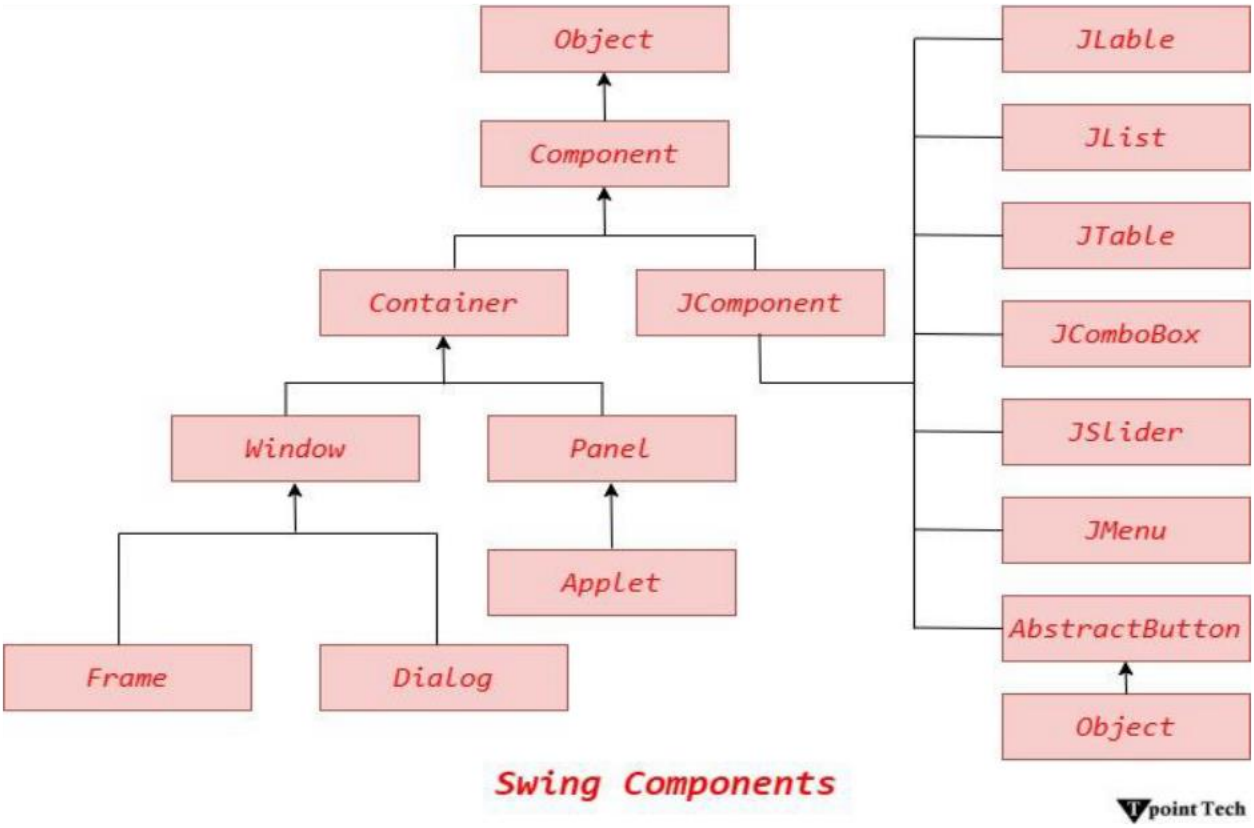
- Provides advanced 2D graphics capabilities (like shapes, text, images, transformations).

5.**Drag and Drop (DnD) API**

- Allows components to be dragged and dropped within or between applications.

Hierarchy of Java Swing classes

The hierarchy of Java swing API is given below.



Java Swing Examples:-

There are two ways to create a frame:

- By creating an object of the Frame class (association)

- By extending the Frame class (inheritance)

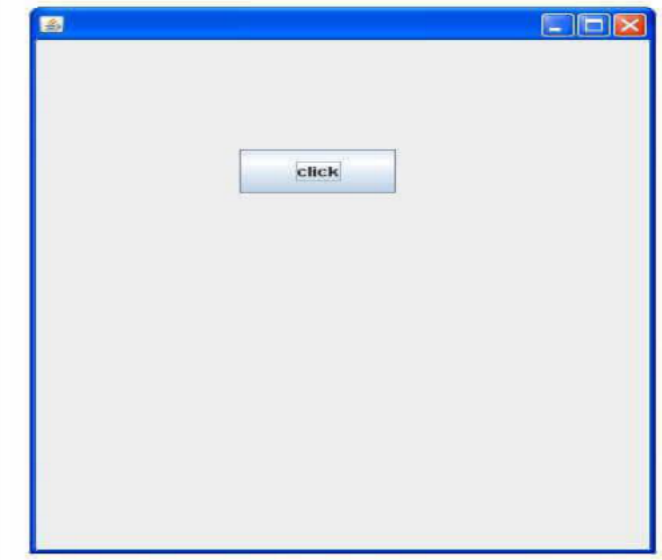
We can write the code of swing inside the main(), constructor or any other method.

PROGRAM: By Creating an Object of the Frame Class

Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

```
import javax.swing.*;

publicclass Main {
publicstaticvoid main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
f.add(b);//adding button in JFrame
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
}
}
```

Output

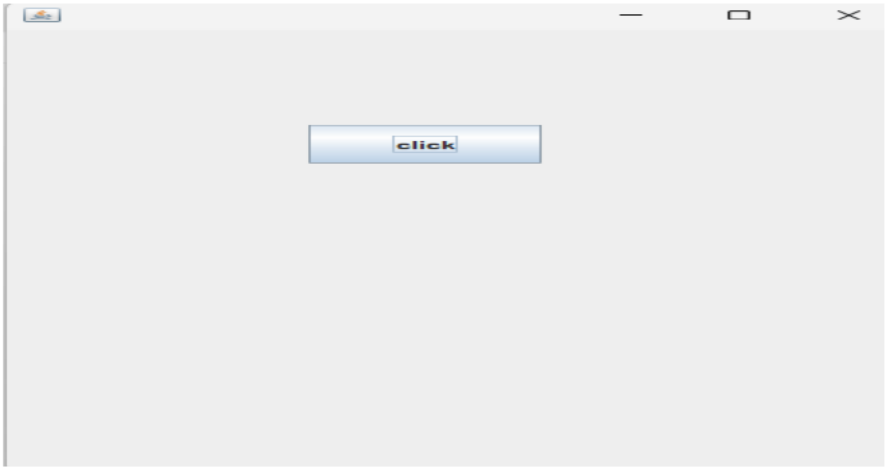


Example of Swing by Association Inside Constructor:-

We can also write all the codes of creating JFrame, JButton and method call inside the Java constructor.

```
import javax.swing.*;

publicclass Main {
JFrame f;
Simple(){
f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");  //creating instance of JButton
b.setBounds(130,100,100, 40);
f.add(b);//adding button in JFrame
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
}
publicstaticvoid main(String[] args) {
new Main();
}
}
```

Output



PROGRAM:- Here's a simple Java Swing program that closes the application when a button is clicked:

CloseApp.java

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

public class CloseApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Close App");
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

        JButton closeButton = new JButton("Close");
        closeButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
        frame.add(closeButton, BorderLayout.CENTER);
        frame.setSize(200, 100);
        frame.setVisible(true);
    }
}
```

How it Works

1. We create a JFrame and set its default close operation to DO_NOTHING_ON_CLOSE.

2. We create a JButton and add an ActionListener to it.

3. When the button is clicked, the System.exit(0) method is called, which closes the application.

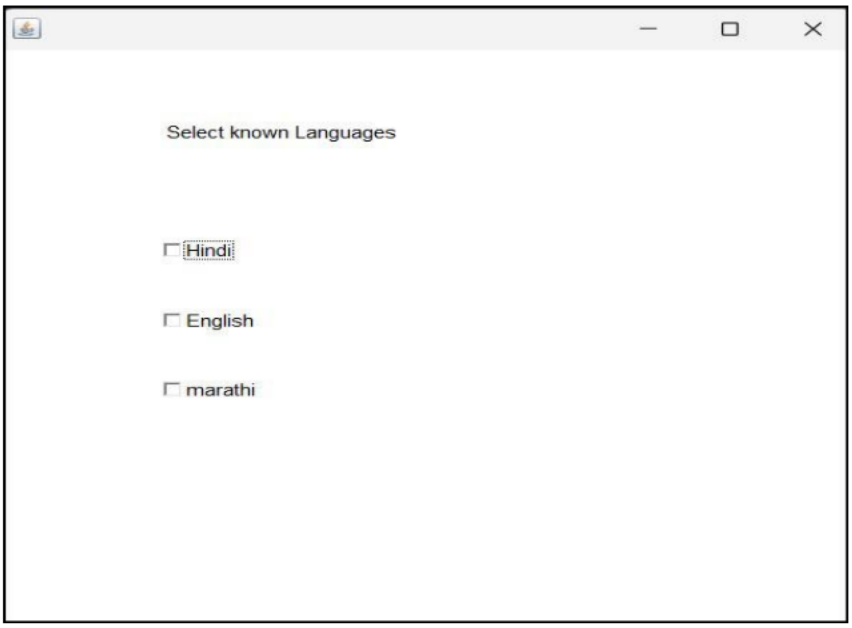
This code demonstrates how to close a Java Swing application programmatically.

PROGRAM:- Program to Add Checkbox in the Frame

```
// Java Swing Program to Add Checkbox
// in the Frame
import java.awt.*;

// Driver Class
class Lan {
    // Main Function
    Lan()
    {
        // Frame Created
        Frame f = new Frame();
        Label l1 = new Label("Select known Languages");
        l1.setBounds(100, 50, 120, 80);
        f.add(l1);

        // CheckBox created
        Checkbox c2 = new Checkbox("Hindi");
        c2.setBounds(100, 150, 50, 50);
        f.add(c2);

        // CheckBox created
```

```
        Checkbox c3 = new Checkbox("English");
        c3.setBounds(100, 200, 80, 50);
        f.add(c3);

        // CheckBox created
        Checkbox c4 = new Checkbox("marathi");
        c4.setBounds(100, 250, 80, 50);
        f.add(c4);
        f.setSize(500, 500);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String ar[]) { new Lan(); }
}
```

Output



Java Swing Using 2 Buttons

```
JButton button1 = new JButton("Add");
JButton button2 = new JButton("Clear");
frame.add(button1);

frame.add(button2);
```

Java Swing Button Group

```
JRadioButton r1 = new JRadioButton("Male");

JRadioButton r2 = new
JRadioButton("Female"); ButtonGroup bg =
new ButtonGroup(); bg.add(r1); bg.add(r2);
```

JProgressBar in Java Swing:-

The JProgressBar is a component in Java Swing used to **visually indicate the progress of a task**, such as file download, data loading, or a long computation.

Example Program: Simulated Task with JProgressBar

```
import javax.swing.*;

import java.awt.*;

public class ProgressBarExample extends JFrame {

    JProgressBar progressBar;

    public ProgressBarExample() {

        setTitle("Progress Bar Example");

        setSize(400, 150);
```

```
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
    setLayout(new FlowLayout());
    progressBar = new JProgressBar(0, 100);
    progressBar.setValue(0);
    progressBar.setStringPainted(true);  // shows percentage text
    add(progressBar);
    setVisible(true);
    runTask();
}
public void runTask() {
    // Simulate task with a background thread
    SwingWorker<Void, Integer> worker = new SwingWorker<>() {
        @Override
        protected Void doInBackground() throws Exception {
            for (int i = 0; i <= 100; i++) {
                Thread.sleep(50); // simulate time-consuming task
                publish(i);      // send value to process()
            }
            return null;
        }
        @Override
        protected void process(java.util.List<Integer> chunks) {
```

```
            int value = chunks.get(chunks.size() - 1);
            progressBar.setValue(value);
        }
        @Override
        protected void done() {
            JOptionPane.showMessageDialog(null, "Task Completed!");
        }
    };
    worker.execute();
  }
  public static void main(String[] args) {
    new ProgressBarExample();
  }
}
```

Explanation:

- JProgressBar is initialized from 0 to 100.
- A SwingWorker is used to simulate a background task.
- As progress increases, setValue(i) updates the bar.
- A message box appears once the task completes.

Output:

- A window with a progress bar filling up from 0% to 100%.
- Once complete, it shows: **"Task Completed!"**

Events in Java Swing:-

In Java Swing, an event is an object that describes a state change in a source. It can be triggered by user interactions such as mouse clicks, key presses, or other actions. Common events include:

ActionEvent: Generated when a button is clicked.

MouseEvent: Generated when a mouse action occurs (click, press, release, enter, exit).

KeyEvent: Generated when a key is pressed, released, or typed.

Event Handling in Java Swing:-

Event handling in Java Swing involves three main components:

Event Source: The component that generates the event (e.g., a button).

Event Object: Encapsulates information about the event (e.g., ActionEvent).

Event Listener: An interface that receives and processes the event (e.g., ActionListener).

Example: Handling a Button Click

```
import javax.swing.*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class ButtonClickExample {

    public static void main(String[] args) {

        // Create a new JFrame

        JFrame frame = new JFrame("Button Click Example");

        frame.setSize(300, 200);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

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

        // Add an ActionListener to the button
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Code to be executed when the button is clicked
                JOptionPane.showMessageDialog(frame, "Button was clicked!");
            }
        });
        // Add the button to the frame
        frame.getContentPane().add(button);
        // Set the frame's visibility to true
        frame.setVisible(true);
    }
}
```

Explanation

1. **Creating the Frame**: A JFrame is created to serve as the main window.
2. **Creating the Button**: A JButton is created with the label "Click Me!".
3. **Adding an Action Listener**: An ActionListener is added to the button. The actionPerformed method is overridden to define what happens when the button is clicked.
4. **Displaying a Message**: When the button is clicked, a message dialog is displayed using JOptionPane.showMessageDialog.

Commonly Used Swing Components:

Component	Description
`JFrame`	Main application window
`JPanel`	Container to group components
`JLabel`	Displays text/images
`JButton`	Button that triggers actions
`JTextField`	Single-line text input
`JTextArea`	Multi-line text input
`JCheckBox`	Checkbox for true/false
`JRadioButton`	Radio button
`JComboBox`	Drop-down list
`JTable`	Displays tabular data
`JScrollPane`	Adds scroll bars to components