JAVA Architecture: (How Java Works? / Working of JVM)

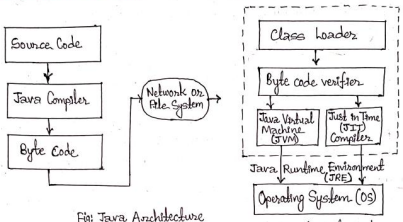


Fig: Java Architecture

→ Java source code (.java files) are compiled into byte code by java compiler.
→ Byte code is not executable code for target machine; rather it's a object code for JVM. Byte code will be stored in class files (.class).
→ Class loader loads all the class files required to execute the program.
→ Byte code verifier checks the byte code and ensure the following:
  ↳ The code follows JVM specifications.
  ↳ There is no unauthorized access to memory.
  ↳ The code doesn't cause any stack overflows.
  ↳ There is no illegal data conversions in code such as float to object references.
→ Once the code is verified, JVM will convert the byte code into machine code.
→ JIT compiler helps the program execution to happen faster. Code is cached by JIT compiler and will be reused for the future needs.
→ Finally with the help of JRE, Operating System runs the code.

#Java Buzzwords: (or Features of Java)

1) **Simple:** Java language is simple because, its syntax is similar to C and C++ and hence easy to learn. After C and C++, Again confusing and rarely used features such as explicit pointers, operator overloading etc, are removed from java.

2) **Object Oriented:** Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour (characteristics). Almost everything in Java is an object.

3) **Distributed:** We can create distributed applications in java. Remote Method Invocation (RMI) and Enterprise Java Beans (EJB) are used for creating distributed applications.

4) **Robust:** Robust simply means strong. Java uses strong memory management. There is automatic garbage collection and lack of pointers that avoids security problem. There is exception handling and type checking mechanism in Java.

5) **Secure:** Java programs run in a virtual machine that protects the underlying operating system from harm. Also, the absence of pointers in Java ensures that programs cannot gain access to memory locations using proper authorization.

6) **Portable:** We can convey java byte code to any platform and execute there. The phrase "write once, run anywhere" is the major concept for the portability. This is because byte code will run on any operating system for which there exists a compatible Virtual Machine.

7) **High Performance:** Java is faster than traditional interpretation since byte code is close to native code. The performance gain is due to caching mechanism of JIT compiler.

8) **Multithreaded:** We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the ....

#Path and Classpath Variables in JAVA:

→ Path is a mediator between developer and operating system to inform binary file path. Classpath is a mediator between developer and compiler to inform library file path those are used in our source code.
→ The path points to the location of JRE, the classpath points to the classes we developed.
→ We don't need to set PATH and CLASSPATH to compile and run Java program while using IDE like Eclipse. Environment variables are required to compile and run java program using CMD.

#Sample JAVA Program:

(We can write this on any text editor like notepad, wordpad etc. to check)

```
class Sample{
    public static void main(String args[]){
        System.out.println("Hello Java");
    }
}
```

Save this file as Sample.java
To Compile: javac Sample.java (write these commands in Using CMD)
To Execute: java Sample
Output: Hello Java

How to get input from user in Java:

Java Scanner class allows the user to take input from the console. It belongs to java.util package.
Syntax: Scanner sc = new Scanner (System.in);
Example:

```
import java.util.*;
class UserInputDemo{
    public static void main(String args[]){
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter a string: ");
        String str=sc.nextLine(); //reads string
        System.out.println("You have entered: "+str);
    }
}
```

#Arrays:

Array is a collection of similar types of elements under same variable name having contigious memory location. Java array is an object that contains elements of similar data type. There are two types of array:

1) **Single Dimensional Array:**
   Syntax: data_type array_name[];
   Example: int arr[];

```
Program:
class TestArray{
    public static void main(String args[]){
        int a[]=new int[5]; //declaration
        a[0]=10; //initialization
        a[1]=20;
        for(int i=0; i<a.length; i++){
            System.out.println(a[i]); //printing array
        }
    }
```

2) **Multidimensional Array:**
   Syntax: data_type array_name[][];
   Example: int arr[3][3]; //i.e, 3 row 3 column.

#For Each Loop:

For each loop provides an alternative approach to traverse the array or collection in JAVA. The advantage is that it eliminates the possibility of bugs and makes code more readable. It always traverses each element one by one. The drawback is it cannot traverse elements in reverse order and cannot have option to skip any element because it does not work on an index basis. But, it is recommended to use because it makes the code readable.

Syntax:
```
for (data_type variable_name : array | collection)
{
    // body of for-each loop
}
```
(array or collection)

Example:
```
class ForEachDemo{
    public static void main(String args[]){
        int a[]={10,11,12,13};
        for(int i:a){
            System.out.println(i);
        }
    }
}
```

#Class and Object: [Imp]

Java is an object-oriented programming language. Everything in Java is associated with classes and objects, along with its attributes and methods. For Example: A bird is a class. Parrot is an object of class bird. Characteristics like color, weight, height etc. are the attributes. How it flys, eats etc. are its methods.

Creating class: Let we create a class called "Bird" as follows:
```
public class Bird{
    int weight=250;
    public void myMethod(){
        System.out.println("Weight is: "+weight+"pounds");
    }
```

Creating object: To create an object of class, we specify class name, followed by object name and use the keyword new. We can create any number of objects once we create a class. Then the attributes and methods of class can be accessed using dot(.) operator. Let we create an object for above created class Bird:

```
Bird myObj=new Bird();
```

Now we can access our class attributes and methods of Bird using myObj as follows:

```
System.out.println(myObj.weight);
```

#Overloading:

In Java, we can define different methods with the same name but either with different number of parameters or with different type of parameters, which is called method overloading. This is one of the examples of polymorphism. In this case the return type of method does not matter.

Example:
```
class Overloading{
    public static void main(String args[]){
        System.out.println(sum(5,7));
        System.out.println(sum(5,7,2));
        System.out.println(sum(5.2f,7));
        System.out.println(sum(5.3,7.4));
    }
    static int sum(int a, int b) {
        return a+b;
    }
    static int sum(int a, int b, int c){
        return a+b+c;
    }
    static float sum(float a, int b){
        return a+b;
    }
    static double sum(double a, double b){
        return a+b;
    }
}
```

#Access Privileges/Access Modifiers:

There are four types of Java access modifiers:

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

#Interface in Java: [Imp]

An interface in Java is a blueprint of a class. It has static constants and abstract methods. The interface in Java is a mechanism to achieve abstraction. There are mainly three reasons to use interface:
→ It is used to achieve abstraction.
→ By interface, we can support the functionality of multiple inheritance.
→ It can be used to achieve loose coupling.

An interface is declared by using interface keyword. All the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:
```
interface interface_name{
    //declare constant fields
    //declare methods that abstract
    // by default.
}
```

#Inner Class:

Inner class refers to the class that is declared inside class or interface, which were introduced, to sum up, same logically relatable classes. Following are certain advantages associated with inner classes:
→ Making code clean and readable.
→ Optimizing the code module.
→ Private methods of the outer class can be accessed, so bringing a new dimension and making it closer to real world.

We use mainly inner class, where we want certain operations to be performed, granting access to limited classes. There are basically four types of inner classes in java:
→ Nested Inner Classes
→ Method Local Inner Classes
→ Static Nested Classes
→ Anonymous Inner Classes.

#Final and Static Modifiers:

Java provides a number of non-access modifiers to achieve many functionalities. Among them, Final and Static modifiers are two main non-access modifiers.
→ The static modifier for creating class methods and variables.
→ The final modifier for finalizing the implementations of classes, methods and variables.

Static Modifier:

Static Variables: The static keyword is used to create variables that will exist independently of any instances created for class. Only one copy of the static variable exists regardless of the number of instances of class.

Static Methods: The static keyword is used to create methods that will exist independently of any instances created for class. Class variables and methods can be accessed using the class name, followed by a dot and the name of variable or method.

Final Modifier:

Final Variables: A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.

Final Methods: A final method cannot be overridden by any subclasses. The final modifier prevents a method from being modified in a subclass.

#Packages: [Imp]

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form: built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing etc.

Advantages of Java Package:
→ Used to categorize classes and interfaces so that they can be easily maintained.
→ It provides access protection.
→ It removes naming collision.

How to compile java package:
Syntax: javac -d directory javafilename
Example: javac -d .Sample.java
where, -d specifies destination to put generated class file.

Simple example of java package:
```
//save as Sample.java
package mypack;
public class Sample{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

#Inheritance: [Imp]

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviours of a parent object. The idea behind inheritance is that we can create new classes based upon existing classes. When we inherit from an existing class, we can reuse methods and fields of the parent class. We can add new methods and fields in our current class also.

Why use inheritance:
→ For Method Overriding.
→ For Code Reusability.

Syntax:
```
class subClass-name extends superClass-name
{
    //methods and fields
}
```

#Overriding:

If subclass has the same method as declared in parent class, it is known as method overriding.

Why use overriding:
→ To provide the specific implementation of a method which is already provided by its superclass.
→ Used for runtime polymorphism.

Rules for Java Method Overriding:
→ The method must have the same name as in the parent class.
→ The method must have same parameter as in the parent class.
→ There must be parent-child (IS-A) relationship (i.e, inheritance).

Note: A static method cannot be overridden. Main method can not be overridden because it is also a static method.

#Handling Exceptions: [Very Imp]

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained. It handles runtime errors such as ClassNotFoundException, IOException, SQLException etc. There are 5 keywords which are used in handling exceptions in Java:

try: The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch: The "catch" block is used to handle the exception. It must be preceded by try block. It can be followed by finally block later.
finally: The "finally" block is used to execute the important code of the program. It is executed whether an exception handled or not.
throw: The "throw" keyword is used to throw an exception.
throws: The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling ...

```
public class ExceptionHandlingExample {
  public static void main(String[] args) {
    final int number = -5;
    try {if (number <= 0) {
      throw new Exception("Number must be greater than zero.");
        }
        System.out.println("Square of the number: " + (number
* number));
    } catch (Exception e) {
        System.out.println("Caught Exception: " +
e.getMessage());
    } finally {
        System.out.println("Finally block always executes.");
    }

    System.out.println("Program ends.");
  }
}
```

Q. Which exception should be declared?
Ans: There are three types of exceptions: checked, unchecked and error. The checked exception should only be declared. Since unchecked exception is under our control so we can correct our code. and error exception is beyond our control eg. we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

throw	throws
i) Java throw keyword is used to explicitly throw an exception.	i) Java throws keyword is used to declare an exception.
ii) Checked exception cannot be propagated using throw only.	ii) Checked exception can be propagated with throws.
iii) Throw is followed by an instance.	iii) Throws is followed by class.
iv) Throw is used within the method	iv) Throws is used with the method signature.
v) We cannot throw multiple exceptions.	v) We can declare multiple exceptions.

Creating Exception Class: (Custom exception Or User-defined exception)

Java platform provides a lot of exception classes that we can use. If we need an exception type that isn't represented by those in the Java platform, we can write one of our own. When we create custom exceptions in Java, we extend either Exception class or RuntimeException class.

Example:

```
class InvalidAgeException extends Exception{
    InvalidAgeException(String s){
        super(s);
    }
}
class TestCustomException1{
    static void validate(int age)throws InvalidAgeException{
    if(age<18)
        throw new InvalidAgeException("not valid");
    else
        System.out.println("welcome to vote");
    }
    public static void main (String args[]){
      try{
        validate(13);
      } catch (Exception m){System.out.println("Exception occured: "+m);}
      System.out.println ("rest of the code...");
    }
}
```

Output: Exception occured: InvalidAgeException: not valid
rest of the code...

(If age given below 18 as input If greater given then welcome to vote will be output)

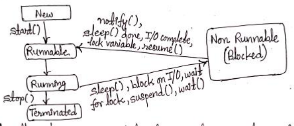
Concurrency:

Concurrency is the ability to run several or multi programs or applications in parallel. The backbone of Java concurrency is threads (A lightweight process, which has it's own files and stacks and can access the shared data from other threads in same process). Java Concurrency package covers concurrency, multithreading, and parallelism on the Java platform. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize CPU time.

7.

Thread States (Life cycle of a Thread):

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution. Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area. The life cycle of thread in java is controlled by JVM. The java thread states are as follows:



New: The thread is in new state if we create an instance of Thread class but before the invocation of start() method.

Runnable: The thread is in runnable state after invocation of start() method, but the thread schedular has not selected it to be the running thread.

Non-Runnable: This is the state when the thread is still alive, but is currently not eligible to run.

Terminated: A thread is in terminated or dead state when its run() method exists.

Writing Multithreaded Programs (Creating OR Multithreaded Programs): [Imp]

There are two ways to create a thread:

i) **By extending Thread class:** Here we need to create a new class that extends the Thread class. The class should override the run() method which is the entry point for the new thread. Then we need to call start() method to start the execution of a thread.

Example:

```
class Multi extends Thread{
    public void run(){
        System.out.println("thread is running...");
    }
    public static void main(String args[]){
        Multi t1=new Multi();
        t1.start();
    }
}
```

ii) **By implementing Runnable interface:** Here we need to give the definition of run() method. This run method is the entry point for the thread and thread will be alive till run method finishes its execution. Once the thread is created it will start running when start() method gets called.

Example:

```
class Multi3 implements Runnable{
public void run(){
    System.out.println("thread is running...");
}
public static void main (String args[]){
    Multi3 m1 = new Multi3();
    Thread t1 = new Thread(m1);
    t1.start();
  }
}
```

Thread Properties:

We can get and set values of most of the thread properties. Major properties of Java threads are: Id, Name, Priority, currentThread, isAlive etc. Some of the thread methods are static and others are non-static. Static methods can be directly invoked through Thread class. But non-static methods need to be invoked by using object of Thread class. Following are important methods available in Thread class that can be used for getting and setting values of thread properties:

Public final void setName (String name): Changes name of the Thread object. There is getName() method for retriving the name.

Public final void setPriority(int priority): Sets the priority of this Thread object. The possible values are between 1 and 10. There is getPriority() method for retriving the priority.

Public final boolean isAlive(): Returns true if thread is alive.

Public static void yeild(): Causes the currently running thread to yield to any other threads of same priority.

Long getId(): This method returns the identifier of this thread.

Void join(): Waits for this thread to die.

Thread Synchronization: [Imp]

Threads in a program are in the same process memory, and therefore have access to the same memory. The biggest problem of allowing multiple threads sharing the same data set is that one operation in one thread could collide with another operation in other threads on the same data. When this happens, the result is un-desirable.

Synchronization is the process of getting lock on the object so that only one thread can access that object. A running thread will enter either a blocked/waiting state when it tries to get the locked object. We can implement synchronization in two ways:

i) **Method level synchronization:** To acquire the object's lock, just call a method that has been modified with the synchronized keyword.

ii) **Block level synchronization:** We can use synchronized block when when we want to block or synchronize only some part of method. Using this we can lock third party objects.

Thread Priorities:

We can set priorities of java threads. Java provides setPriority() and getPriority() methods for setting and reading priorities of threads. Value of priority can range from 1-10. Default value of priority is 5. High priority threads get higher chances of execution from JVM. However, exact behaviour depends upon JVM.

Example:

```
class ThreadDemo extends Thread{
        String tname;
        ThreadDemo(String n){
                tname=n;
        }
}
public class ThreadPriority{
    public static void main(String args[]){
        ThreadDemo x = new ThreadDemo("T1");
        ThreadDemo y = new ThreadDemo("T2");
        //Displaying priority of threads or getting priority
        System.out.println("Priority of x: "+x.getPriority());
        //Setting priority of threads and displaying again
        x.setPriority(8);
        System.out.println("Priority of x: "+x.getPriority());
    }
}
```

Working with Files:

Java I/O (Input and Output) is used to process the input and produce the output based on input. Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations. A stream is sequence of data. In java stream is composed of bytes. In java, 3 streams are created for us automatically. All these streams are attached with console.

System.out → Standard output stream
System.in → Standard input stream
System.err → Standard error stream

Byte Stream Classes:

Byte streams process data byte by byte (8 bits). For example FileInputStream is used to read from source and FileOutputStream to write to the destination. Following is an example:

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

Character Stream Classes: [Imp]

Or, Write a simple java program that reads data from one file and writes to another file

(Question may be asked like this. → Write character stream class example in ans:)

In Java, characters are stored using Unicode conventions. Character stream automatically allows us to read/write data character by character. For example FileReader and FileWriter are character streams used to read from source and write to destination.

Example:

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

Random Access File Class:

This class is used for reading and writing to random access file. A random access file behaves like a large array of bytes. There is a cursor implied to the array called pointer, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is thrown. It is a type of IOException.

Constructor	Description
Random Access File (File file, String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
Random Access File (String name, String mode)	Creates a random access file stream to read from and optionally to write to, a file with the specified name.

Multiple Inheritance with Interface: [Imp]

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Java does not support multiple inheritances with classes. In java, we can achieve multiple inheritances only through Interfaces. An interface is declared by using interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface.

Multiple inheritance can be achieved with interfaces, because the class can implement multiple interfaces. To implement multiple interfaces, seperate them with a comma. In the following example class Animal is derived from interface AnimalEat and AnimalTravel.

```
interface AnimalEat{
        void eat();
}
interface AnimalTravel{
        void travel();
}

class Animal implements AnimalEat, AnimalTravel{
    public void eat(){
            System.out.println("Animal is eating");
    }
    public void travel(){
            System.out.println("Animal is traveling");
    }
}
```

Concept of AWT:

Java AWT (Abstract Windowing Toolkit) is an API to develop GUI or window-based application in java. Java AWT components are platform-dependent i.e, components are displayed according to the view of operating system. AWT calls Operating Systems subroutine for creating components such as textbox, checkbox, button etc. For example if we are creating a textbox in AWT that means we are actually asking OS to create a textbox for us. This is the reason why AWT components look different on different operating systems. An application build on AWT would look like a windows application when it runs on Windows, but the same application would look like a Mac application when runs on Mac OS.

Java AWT vs. Java Swing: [Imp]

Java AWT	Java Swing
i) AWT components are platform-dependent.	i) Java Swing components are platform independent.
ii) AWT components are heavyweight.	ii) Swing components are lightweight.
iii) AWT does not support pluggable look and feel.	iii) Swing supports pluggable look and feel.
iv) AWT does not follow MVC.	iv) Swing follows MVC.
v) AWT provides less components than swing.	v) Swing provides more powerful components such as tables, lists, scrollpanes etc.

Java Applets:

→ Applets are small Java applications which can be accessed on an Internet server, transported over the internet, and can be installed and run automatically as a part of a web document.

→ The applet can create a GUI and it has restracted access to resources so that complicated computations can be carried out without adding the danger of viruses and violating data integrity.

→ Any java applet is a class that extends the class of java.applet.Applet.

→ There is no main() method in an Applet class. The JVM can operate an applet application using either a web-browser plug-in or a distinct runtime environment.

Example of Simple Applet:

```
import java.awt.*;
import java.applet.*;
public class Simple extends Applet
{
  public void paint (Graphics g){
      g.drawString("A simple Applet", 20,20);
  }
}
```

Benefits of Applets:

→ As it operates on the client side, it requires much less response time.
→ Any browser that has JVM operating in it can operate it.

Applet Life Cycle:

These 4 methods are overridden by most applets. These 4 methods are the lifecycle of the Applet.

init(): The first technique to be called is init(). This is where we initialize the variable. This is called only once during applet runtime.

start(): Method start() is called after init(). This technique is called after it has been stopped to restart an applet.

stop(): Method stop() is called to suspend threads that do not need to operate when the applet is not noticeable.

destroy(): The destroy() method is called if we need to remove our applet from memory entirely.

Swing Class Hierarchy:

→ Java Swing 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.

→ Unlike AWT, Java Swing provides platform-independent and lightweight components.

→ The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JMenu, JColorChooser etc.

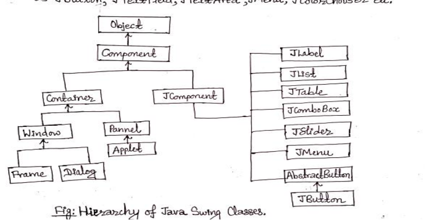


Fig: Hierarchy of Java Swing Classes.

#Components and Containers:

Components: At the top of AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a component. All user interface elements that are displayed on screen and that interact with the user are subclasses of component. It defines over a hundreds of public methods that are responsible for managing events. Component object is responsible for remembering the current foreground and background colors, currently selected text font, and current text alignment.

Containers: The Container class is a subclass of Component. It has additional features that allow other Component objects to be placed within it. Other container objects can also be stored inside of a Container since they are themselves instances of Component. Thus, allowing multilevel containment system. A container is responsible for positioning any component placed on it. It does this through the use of various layout managers. Default layout is present in each container, which can be overridden using setLayout method.

#Layout Managements: [imp]

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes. There are following classes that represents the layout managers:

- → java.awt.BorderLayout
- → java.awt.GridLayout
- → java.awt.GridBagLayout
- → javax.swing.GroupLayout
- → Using No Layout Managers
- → Custom Layout Managers

No Layout (Absolute Positioning):

For No layout we need to import javax.swing.* package necessary to access JFrame, JLabel, and JTextField class. User Interface of having the lookAndFeelDecorated UI is set to true as below:

```
JFrame.setDefaultLookAndFeelDecorated(true);
```

Then we need to initialize variables in our Main, variable frame as JFrame, label as JLabel, and textField as JTextField.

```
JLabel label = new JLabel("Name:");
JTextField textField = new JTextField("Hello", 15);
```

To set the layout without a layout we will use null keyword in the setLayout method of the frame.

```
frame.getContentPane().setLayout(null);
```

Now we will proceed with the absolute positioning of our components with the use of the setBounds method.

```
label.setBounds(20,20,200,40);
```

Flow layout

The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). Fields of FlowLayout class are Left, right, center, leading, trailing etc. and specified as:

```
public static final int CENTER
```

(only values are changed here: LEFT, RIGHT etc.)

FlowLayout(int align, int hgap, int vgap): creates a flow layout with the given alignments, horizontal gap, and vertical gap.

```
Import javax.swing.*;
import java.awt.*;
public class FlowLayoutExample {
   public static void main(String[] args) {
      JFrame frame = new JFrame("FlowLayout Example");
      frame.setLayout(new FlowLayout());
      frame.add(new JButton("Button 1"));
      frame.add(new JButton("Button 2"));
      frame.add(new JButton("Button 3"));
      frame.setSize(300, 200);
      frame.setVisible(true);
   }
}
```

Border Layout

A border layout places components in upto five areas: top, bottom, left, right, and center. All extra space is placed in the center area. Tool bars that are created using JToolBar must be created with a BorderLayout container, if we want to be able to drag and drop the bars away from their starting positions.

Constructor or Method	Purpose
BorderLayout(int horizontalGap, int verticalGap)	Defines a border layout with specified gaps between components.
setHgap(int)	Sets the horizontal gap between components.
setVgap(int)	Sets the vertical gap between components.

Example:

```
import java.awt.*;
import java.swing.*;
public class Border{
    JFrame f;
    Border(){ f=new JFrame();
        JButton b1=new JButton("Border");
        f.add(b1, BorderLayout.CENTER);
        f.setSize(300,300);
        f.setVisible(true);
    }
    public static void main(String args[]){
        new Border();
    }
}
```

(create and add multiple buttons; for default view notes alternative provided above)

GridLayout:

GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns. The constructors used are:

```
GridLayout(int rows, int cols)
GridLayout(int rows, int cols, int hgap, int vgap)
```

Example:

```
import java.awt.*;
import java.swing.*;
public class MyGridLayout{
    JFrame f;
    MyGridLayout(){
        f=new JFrame();
        JButton b1=new JButton("1");
        f.add(b1);
        f.setLayout(new GridLayout(3,3));
        f.setSize(300,300);
        f.setVisible(true);
    }
    public static void main(String args[]){
        new MyGridLayout();
    }
}
```

(Create multiple buttons 1-8; add multiple buttons to frame; Smaller example. Only this line changes setLayout line.)

GridBag Layout:

GridBagLayout is a sophisticated, flexible layout manager. It aligns components to span more than one cell. The rows in the grid can have different heights, and grid columns can have different widths.

```
JPanel pane = new JPanel(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
//For each component to be added to this container:
//Create the component...
//Set instance variables on the GridBagConstraints instance...
pane.add(theComponent, c);
```

Group Layout:

GroupLayout is a layout manager that was developed for use by GUI builder tools, but it can also be used manually. GroupLayout works with the horizontal and vertical layouts seperately. The layout is defined for each dimension seperately.

```
GroupLayout layout = new GroupLayout(panel);
panel.setLayout(layout);
```

We specify automatic gap insertion:

```
layout.setAutoCreateGaps(true);
layout.setAutoCreateContainerGaps(true);
```

#GUI Controls:

1) Text Input:

Text input contains Text Fields, Password Fields, Text areas, Scroll pane, Label and Labeling Components.

Text Field: A text field is a basic text control that enables the user to type a small amount of text. When the user indicates that text entry is complete (usually by pressing Enter), the text field fires an action event.

```
Syntax: textField = new JTextField(20);
```

TextArea: To obtain long texts like paragraph which are more than one line of input, we use text area. TextField only takes single line of text while TextArea takes multiple lines of text. It also allows user to edit the text.

```
Syntax: textArea = new JTextArea(5,20);
JScrollPane = new JScrollPane(textArea);
textArea.setEditable(false);
```

(for getting scroll option on textArea; set true if we want editable)

Password Field: A password field provides specialized text fields for password entry. For security reasons, a password field does not show characters that the user types, instead displays different from typed such as an asterisk (*). A password field stores its value as an array of characters, rather than a string.

```
Syntax: passwordField = new JPasswordField(10);
passwordField.setActionCommand(OK);
passwordField.addActionListener(this);
```

ScrollPane: ScrollPane is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component.

```
Syntax: JScrollPane scrollableTextArea = new JScrollPane(textArea);
```

Label: Label is used to display a single line of read only text. The text can be changed by a programmer but a user cannot edit it directly. To create a label, we need to create the object of Label class.

```
Syntax: l1 = new Label("First");
l1.setBounds(50,100,100,50);
```

2) Choice Components:

Choice components include Check Boxes, Radio Buttons, Borders, Combo Boxes, Sliders.

Check Boxes: Check box is used to turn an option true or false. Clicking on a check box changes its state from true to false or false to true.

```
Syntax: JCheckBox checkbox = new JCheckBox("Married");
checkbox.setBounds(100,100,50,50);
```

Radio Button: It is used to choose one option from multiple options. It is widely used in exam systems or quiz. It should be added in ButtonGroup to select one radio button only.

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

Borders: Border component is used to place border in our components.

```
Syntax: Border bdr = new LineBorder(Color.ORANGE, 4, true);
```

Combo Box: The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of menu.

```
Syntax: String country[]={"Nepal", "India", "Aus", "U.S.A"};
JComboBox cb = new JComboBox(country);
```

Sliders: The Java JSlider class is used to create the slider. By using JSlider, a user can select a value from a specific range.

```
Syntax: JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
```

#Menus:

```
//Create the menu bar.
menuBar = new JMenuBar();
//Build the first menu.
menu = new JMenu("A Menu");
menuBar.add(menu);
//A group of JMenuItems
menuItem = new JMenuItem("Both text and icon", new ImageIcon("
        images/myImg.jpg"));
. . . .
menu.addSeparator();
```

Java JMenuItem and JMenu Example:

```
import javax.swing.*;
class MenuExample{
    JMenu menu, submenu;
    JMenuItem i1, i2, i3, i4;
    MenuExample(){
        JFrame f = new JFrame("Menu and MenuItem Example");
        JMenuBar mb = new JMenuBar();
        menu = new JMenu("Menu");
        submenu = new JMenu("Sub Menu");
        i1 = new JMenuItem("Item 1");
        i2 = new JMenuItem("Item 2");
        i3 = new JMenuItem("Item 3");
        i4 = new JMenuItem("Item 4");
        submenu.add(i3);
        submenu.add(i4);
        menu.add(submenu);
        mb.add(menu);
        f.setJMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String args[]){
        new MenuExample();
    }
}
```

Icons in Menu Items:

Icons in menu items can be added as:

```
Icon myIcon = new ImageIcon("Resources/myIcon.png");
```

Enabling and Disabling Menu Items:

```
//For disabling
menuItem1.setEnabled(false);
//For enabling
menuItem1.setEnabled(true);
```

Tooltips:

We can create a tooltip for any JComponent with setToolTipText() method. For example, to add tooltip to PasswordField we do as following:

```
field.setToolTipText("Enter your Password");
```

Check Box in Menu Items: & RadioButton in Menu Items:

JCheckBoxMenuItem class represents checkbox which can be included on a menu. A CheckBoxMenuItem can have text or graphic icon or both, associated with it. MenuItem can be selected or deselected. MenuItems can be configured and controlled by actions.

```
Example: JMenu fileMenu = new JMenu("File");
JCheckBoxMenuItem caseMenuItem = new JCheckBoxMenuItem("Option1");
fileMenu.add(caseMenuItem);
```

#Similarly JRadioButtonMenuItem class represents RadioButton which can be included on a menu similarly as we did for checkbox.

Pop-up Menu:

PopupMenu can be dynamically popped up at specific position within a component. It inherits the Menu class.

```
AWT PopupMenu class declaration
public class PopupMenu extends Menu implements MenuContainer,Accessible
```

Example:

```
import java.awt.*;
import java.awt.event.*;
class PopupMenuExample{
PopupMenuExample(){
  final Frame f=new Frame("PopupMenu Example");
  final PopupMenu popupmenu=new PopupMenu("Edit");
    MenuItem copy=new MenuItem("Copy");
    copy.setActionCommand("Copy");
    MenuItem paste=new MenuItem("Paste");
    copy.setActionCommand("Paste");
    popupmenu.add(copy);
    popupmenu.add(paste);
}
  f.add(popupmenu);
  f.setSize(400,400);
  f.setLayout(null);
  f.setVisible(true);
public static void main(String args[]){
        new PopupMenuExample();
}
}
```

Keyboard Mnemonics and Accelerators:

A mnemonic is a key-press that opens a JMenu or selects a MenuItem when the menu is opened. An accelerator is a key-press that selects an option within the menu without it ever being open. The purpose of all this is to let people who really know the program to access functions quickly, and let people that don't use a mouse (some don't) to access the MenuBar.

```
Example: JMenu menu = new JMenu("Menu");  //Create Menu
    menu.setMnemonic('M');  //Set Mnemonic
    JMenuItem menuItem = new JMenuItem("Item");
    menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_I,
                            KeyEvent.SHIFT_MASK));
    menu.add(menuItem);
```

Toolbars:

JToolBar container allows us to group other components, usually buttons with icons in a row or column. JToolBar provides a component which is useful for displaying commonly used actions or controls.

```
Example: JToolBar toolbar = new JToolBar();
    toolbar.setRollover(true);
    toolbar.add(new JButton("Edit"));
    toolbar.add(new JComboBox(new String[]{"option1", "option2"}));
    Container contentPane = myframe.getContentPane();
    contentPane.add(toolbar, BorderLayout.NORTH);
    myframe.setSize(450,250);
    myframe.setVisible(true);
```

Option Dialogs: & Creating Dialogs:

The JOptionPane class is used to provide standard dialog boxes such as message dialog box, confirm dialog box and input dialog box. These dialog boxes are used to display information or get input from the user. The JOptionPane class inherits JComponent class.

```
JOptionPane class declaration:
    public class JOptionPane extends JComponent implements Accessible
```

```
Example: showMessageDialog():
  import javax.swing.*;
  public class OptionPaneExample{
      JFrame f;
      OptionPaneExample(){
              f=new JFrame();
              JOptionPane.showMessageDialog(f,"Hello");
      }
  public static void main(String args[]){
        new OptionPaneExample();
  }
}
```

RMI	CORBA
i) RMI is a Java-specific technology.	i) CORBA is independent of programming languages.
ii) It uses Java interface for implementation.	ii) It uses Interface Definition language (IDL) to seperate interface from implementation.
iii) RMI programs can download new classes from remote JVMs.	iii) CORBA doesn't have this code sharing mechanism.
iv) RMI passes objects by remote reference or by value.	iv) CORBA passes objects by reference.
v) Distributed garbage collection is available.	v) Distributed garbage collection is not available.
vi) Generally simpler to use	vi) More complicated.
vii) It is free of cost	vii) Cost money according to the vendor.

File Choosers & Color Choosers:

The object of JFileChooser class represents a dialog window from which the user can select file. Similarly the object of JColorChooser class represents a dialog window from the user which the user can select color. They both inherit from JComponent class.

Example: File Choosers:

```
import javax.swing.JFileChooser;
import java.io.File;
public class FileChooserExample(){
    JFileChooser fileChooser=new JFileChooser();
    fileChooser.setCurrentDirectory(new File(System.getProperty("user.home")));
    int result= fileChooser.showOpenDialog(parent);
    if (result==JFileChooser.APPROVE_OPTION){
        //user selects a file
    }
    File selectedFile=fileChooser.getSelectedFile();
    System.out.println("Selected file:"+selectedFile.getAbsolutePath());
}
public static void main(String args[]){
        new FileChooserExample();
}
```

Example: Color Choosers:

```
import javax.swing.*;
public class ColorChooserExample extends JFrame implements ActionListener{
    JButton b;
    Container c;
    ColorChooserExample(){
        c.getContentPane();
        c.setLayout(new FlowLayout());
        b=new JButton("color");
        b.addActionListener(this);
        c.add(b);
    }
public void actionPerformed(ActionEvent e){
    Color initialcolor=Color.RED;
    Color color=JColorChooser.showDialog(this,"Select a color",initialcolor);
    c.setBackground(color);
}
public static void main(String args[]){
    ColorChooserExample ch=new ColorChooserExample();
    ch.setSize(400,400);
    ch.setVisible(true);
    ch.setDefaultCloseOperation(EXIT_ON_CLOSE);
}
}
```

Internal Frames:

JInternalFrame is a part of Java Swing. JInternalFrame is a container that provides many features of a frame which includes displaying title, opening, closing, resizing, support for menu bar etc.

Example:

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class solution extends JFrame{
    static JFrame f; //frame
    static JLabel l; //label to display text
    public static void main(String args[]){
        f=new JFrame("frame"); //Create new frame
        JInternalFrame in=new JInternalFrame();
        in.setTitle("Internal Frame");
        JButton b=new JButton("button");
        l=new JLabel("this is a JInternal Frame");
        JPanel p=new JPanel();
    //add label and button to pannel
        p.add(l); p.add(b);
        in.setVisible(true);
        in.add(p);
        f.add(in);
        f.setSize(300,300);
        f.show();
    }
}
```

Advance Swing Components:

1) **List:** A JList presents the user with a group of items, displayed in one or more columns, to chose from. Lists can have many items, so they are often put in scroll panes.

```
lst=new JList(data);
lst.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
lst.setLayoutOrientation(JList.HORIZONTAL_WRAP);
lst.setVisibleRowCount(-1);
```

2) **Trees:** With the JTree class, we can display hierarchical data. A JTree object does not actually contain our data, it simply provides a view of data. Like any non-trivial Swing component, the tree gets data by quering its data model.

```
private JTree tree;
...
public TreeDemo(){
    DefaultMutableTreeNode top = new DefaultMutableTreeNode("Java");
    createNodes(top);
    tree = new JTree(top);
...
}
```

3) **Tables:** The JTable class is used to display data in tabular form. It is composed of rows and columns.

Example:

```
import javax.swing.*;
public class TableExample{
    JFrame f;
    TableExample(){
        f=new JFrame();
        String data[][]={{"101","Amit","670000"},
                         {"102","Jai","650000"}};
        String column[]={"ID","Name","SALARY"};
        JTable jt=new JTable(data,column);
        jt.setBounds(30,40,200,300);
        JScrollBar sp=new JScrollPane(jt);
        f.add(sp);
        f.setSize(300,400);
        f.setVisible(true);
    }
public static void main(String args[]){
    new TableExample();
}
}
```

Registeration Methods:

- For Button or MenuItem:
  - ⇒ public void addActionListener(ActionListener a){ }
- For Checkbox or Choice:
  - ⇒ public void addItemListener(ItemListener a){ }
- For TextArea:
  - ⇒ public void addTextListener(TextListener a){ }
- For TextField:
  - ⇒ public void addActionListener(ActionListener a){ }
  - ⇒ public void addTextListener(TextListener a){ }
- For List:
  - ⇒ public void addActionListener(ActionListener a){ }
  - ⇒ public void addItemListener(ItemListener a){ }

Sum of two nums using swing

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class SimpleCalculator {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Simple Calculator");
        frame.setSize(300, 150);
        frame.setLayout(new FlowLayout());
        JTextField n1 = new JTextField(10);
        JTextField n2 = new JTextField(10);
        JButton b = new JButton("Calculate");
        JLabel l = new JLabel("Sum:");
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int num1 = Integer.parseInt(n1.getText());
                int num2 = Integer.parseInt(n2.getText());
                int sum = num1 + num2;
                l.setText("Sum: " + sum);
            }});
        frame.add(new JLabel("Number 1:"));
        frame.add(n1);
        frame.add(new JLabel("Number 2:"));
        frame.add(n2);
        frame.add(b);
        frame.add(l);
        frame.setVisible(true);
    }}
```

Sum and difference swing and adapter

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
public class SumDifferenceCalculator {
    public static void main(String[] args) {
        JFrame f = new JFrame("Sum and Difference ");
        f.setSize(300, 200);
        f.setLayout(new FlowLayout());
        JTextField a = new JTextField(10);
        JTextField b = new JTextField(10);
        JLabel r = new JLabel("Result: ");
        MouseAdapter m = new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                int p = Integer.parseInt(a.getText());
                int q = Integer.parseInt(b.getText());
                int s = p + q;
                r.setText("Sum: " + s);
            }
            public void mouseReleased(MouseEvent e) {
                int p = Integer.parseInt(a.getText());
                int q = Integer.parseInt(b.getText());
                int d = p - q;
                r.setText("Difference: " + d);
            }
        };
        f.addMouseListener(m);
        f.add(new JLabel("Number 1: "));
        f.add(a);
        f.add(new JLabel("Number 2: "));
        f.add(b);
        f.add(r);
        f.setVisible(true);
    }}
```

Event Handling Concepts: [Imp]

(Why we need event handling, how event handling works or describe delegation event model?)

An event can be defined as changing the state of an object or behaviour by performing actions. Actions can be a button click, cursor movement, mouse clicks, keypress through keyboard, page scrolling etc. Events are of two types:

1) **Foreground Events:** Events that require user interaction to generate are foreground events. Example: clicking a button, cursor movements etc.

2) **Background Events:** Events that don't require user interaction to generate are background events. Example: operating system failures/interrupts, operation completion etc.

Event Handling: It is a mechanism to control the events and to decide what should happen after an event occur. To handle the events, java follows Delegation Event Model. It has Sources and Listeners:

Source: Events are generated from the source, there are various sources like buttons, checkboxes, list, menu-item etc. to generate events.

Listeners: Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

To perform Event Handling, we need to register the Source with the Listener.

Syntax: addTypeListener()

where type represents the type of event.

For Example: For KeyEvent we use addKeyListener() to register. Similarly For ActionEvent we use addActionListener() to register.

Adapter Classes: [Imp]

(At least concept & adapter class diag)

Java adapter classes provide the default implementation of listener interfaces. If we inherit the adapter class, we will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.

Advantages of using Adapter classes:

- It provides ways to use classes in different ways.
- It increases transparency of classes.
- It provides a way to include related patterns in the class.
- It increases the reusability of the class.

Adapter class	Listener Interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
FocusAdapter	FocusListener
MouseMotionAdapter	MouseMotionListener

Mouse adapter example

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
public class SimpleMouseAdapterExample {
    public static void main(String[] args) {
        MouseAdapter ma= new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                System.out.println("Mouse Clicked");
            }};
    someComponent.addMouseListener(ma);
    }}
```

Action listeners

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class ActionEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Action Event ");
        JButton button = new JButton("Click Me");
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked!");
            }});
        frame.add(button);
        frame.setSize(200, 100);
        frame.setVisible(true);
    }}
```

Key events

```
import javax.swing.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
public class KeyEventExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Key Event Example");
        frame.setSize(300, 200);
        JTextField textField = new JTextField(20);
        textField.addKeyListener(new KeyListener() {
            @Override
            public void keyPressed(KeyEvent e) {
        System.out.println("Key Pressed: +e.getKeyChar());
            }
            @Override
            public void keyReleased(KeyEvent e) {}
            @Override
            public void keyTyped(KeyEvent e) { }
        });
        frame.add(textField);
        frame.setVisible(true);
    }}
```

Event Listener vs. Adapter class: [Imp]

Many of the listener interfaces have more than one method. When we implement a listener interface in any class then we must have to implement all the methods declared in that interface because all the methods in an interface are final and must be override in class which implement it.

Adapter class makes it easy to deal with this situation. An adapter class provides empty implementations of all methods defined by that interface. Adapter classes are very useful if we want to override only some of the methods defined by that interface.

JDBC:

(JDBC introduction asap)

JDBC stands for Java Database Connectivity, which is standard Java API for database-independent connectivity between the Java programming language and a wide range of databases. The JDBC library includes APIs for each of tasks commonly associated with database usage:

- Making a connection to a database
- Creating SQL or MySQL statements
- Executing SQL or MySQL queries in the database
- Viewing and Modifying the resulting records.

JDBC Architecture:

JDBC Architecture consists of two layers:

JDBC API: This provides the application-to-JDBC Manager connection.

JDBC Driver API: This supports the JDBC Manager-to-Driver connection.

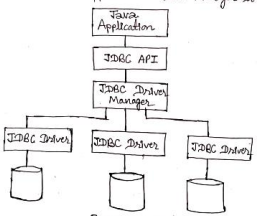


Fig: JDBC Architecture

Common JDBC Components:

The JDBC API provides the following interfaces and classes:

Driver Manager: This class manages a list of database drivers. Match connection requests from java application with proper database driver. Driver: This interface handles the communications with the database server.

Connection: This interface consists all methods for contacting a database. All communication with database is done through connection object only.

Statement: We use objects created from interface to submit the SQL statements to the database.

ResultSet: These objects hold data retrived from database after we execute an SQL query.

SQLException: This handles any errors that occur in database application.

#Benefits of JDBC:

- It enables enterprise applications to continue using existing data even if the data is stored on different database management systems.
- The combination of the java API and the JDBC API makes the database transferable from one vendor to another without modifications in the application code.
- JDBC is cross-platform or platform independent.
- With JDBC it is easy to deploy and economical to maintain.

JDBC Driver Types: [Imp]

There are 4 types of JDBC drivers:

Type1: JDBC – ODBC Bridge:
- → To use a Type 1 driver in a client machine, an ODBC driver should be installed and configured correctly.
- → This driver does not directly interact with the database. To interact, it needs ODBC driver. It converts JDBC method class into ODBC method class.
- → It can be used to connect to any type of databases.

Type2: Native–API driver:
- → Type 2 drivers are written partially in Java and partially in native code.
- → The Native–API of each database should be installed in the client system before accessing a particular database.
- → This driver converts JDBC method calls into native calls of the database API.

Type3: Net Pure Java driver:
- → In type 3 driver, the JDBC driver on the client machine uses the socket to communicate with the middleware at the server.
- → The client database access requests are sent through the network to the middleware.
- → Type-3 drivers are fully written in Java, hence they are portable.

Type4: Pure Java driver:
- → This driver interact directly with database.
- → No client-side or server-side installation.
- → It is fully written in Java, hence they are portable drivers.

JDBC Configuration:

(Steps to connect to database if asked write in short about JDBC configuration, connection, statement & Result set.)

Let we are using mysql database, now to connect Java application with MySQL database, we need to follow following steps:

1. Driver class: The driver class for the mysql database is com.mysql.jdbc.Driver.
2. Connection URL: The connection URL for mysql database is jdbc:mysql://localhost:3306/roshan where jdb is API, mysql is database, localhost is server name on which mysql is running, 3306 is port number and roshan is database name.
3. Username: The default username for the mysql database is root.
4. Password: It is given by user at the time of installing mysql database.

Managing Connections:

Create the connection object with getConnection() method of DriverManager class.

```
Connection con = DriverManager
getConnection(url: "jdbc:mysql://localhost:3306/roshan", user: "root",
    password: "");
```

(i.e., connection url //dburl/dbname; username; password)

Statement:

Create the Statement object with createStatement() method of Connection interface.

```
Statement stmnt = con.createStatement();
```

Result Set:

The executeQuery() method of Statement interface is used to execute queries to database.

```
String select = "SELECT * FROM roshan";
ResultSet rs = stmnt.executeQuery(select);
```

Note:
- i) Use executeQuery(string) method to retrive data from database. It returns ResultSet object.
- ii) Use executeUpdate(string) method to insert, update, delete data in database. It returns int for the number of affected rows.

Close Connection:

The close() method of Connection interface is used to close the connection.

```
con.close();
```

Fetching multiple results:

```
System.out.println("ID \t NAME \t SALARY");
while (rs.next()){
    //rs.next() checks whether next row is available
    System.out.print (rs.getInt(s: "id") + "\t");
    System.out.print (rs.getString(s: "name") + "\t");
    System.out.print (rs.getFloat (s: "salary"));
}
```

(rs is object created by us for ResultSet)

SQL Exceptions:

Exception handling allows us to handle exceptional conditions such as program-defined errors in a controlled fashion to maintain the flow of program and to avoid program from crash. JDBC Exception handling is very similar to Java Exception but for JDBC, the most common exception we will deal with is java.sql.SQLException.

When exception occurs, an object of type SQLException will be passed to catch clause. The passed SQLException object has following methods available for retriving additional information about the exception:

- getErrorCode()
- getMessage()
- getSQLState()
- getNextException()
- printStackTrace()
- etc.

(Very less chance but if example asked write similar example of exception as we write in unit 1 with try, catch including sql query in try block like: sql="Update Employees SET age=?" Catch block should throw output as: "Invalid SQL Statement".)

#scrollable resultset

```
Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM table_name");
```

#updatable resultset

```
Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM table_name");
```

#how to traverse or update

```
rs.next(); // Move to the first row
rs.updateString("column_name", "new_value");
rs.updateRow(); // Commit the changes to the db
```

DDL and DML Operations using Java:

Data Definition language (DDL) and Data Manipulation language (DML) together forms a Database Language. The basic difference between DDL and DML is that DDL is used to specify the database schema data structure. On the other hand, DML is used to access, modify or retrive the data from the database.

DDL query using java:

```
String createTable = "CREATE TABLE" + "test (" + "id INT(10), " +
    "name VARCHAR(20), " + "price FLOAT(20), ")";
stmnt.execute(createTable);
```

DML query using java:

```
String sql = "insert into test(name, address, price) values" + "
    ('Roshan', 'Mahendranagar', 1000000)";
stmnt.executeUpdate(sql);
stmnt.executeQuery(sql);
```

Prepared Statements: [Imp]

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query. The performance of the application will be faster if we use PreparedStatement interface because query is compiled only once. To get the instance of PreparedStatement, the prepareStatement() method of connection interface is used.

Syntax:

```
public PreparedStatement prepareStatement (String query) throws SQLException{}
```

Example of PreparedStatement interface that inserts the record:

```
create table emp(id number(10), name varchar(50));
PreparedStatement stmt = con.prepareStatement ("insert into Emp values(?,?)");
stmt.setInt(1,101);
stmt.setString(2,'Roshan');
int i = stmt.executeUpdate();

rs.setURL("jdbc:mysql://localhost:3306/roshandb");
rs.setUsername("root");
rs.setPassword ("root");

rs.setCommand("select * from teachers");
rs.execute();
```

(con → connection object)

Cached Row Set:

A CachedRowSet object is a container for rows of data that caches it's rows in memory, which makes it possible to operate without keeping the database connection open all the time.

A CachedRowSet object makes use of a connection to the database only briefly: while it is reading data to populate itself with rows, and again while it is committing changes to the underlying database. So the rest of the time, a CachedRowSet object is disconnected, even while its data is being modified. Hence it is called disconnected row set.

JDBC Transactions:

- → A transaction is a set of actions to be carried out as a single, atomic action. Either all of the actions are carried out, or none of them are.
- → The classic example of when transactions are necessary is the example of bank accounts.
- → Let we need to transfer 10,000 rupees from one account to the other.
- → We do so by subtracting Rs 10,000 from the first account, and adding Rs 10,000 to the second account.
- → If this process fails after we have subtracted Rs 10,000 from the first account, the Rs 10,000 are never added to second account.
- → The money is lost in cyber space. To solve this addition and subtraction are grouped into a transaction.
- → If the subtraction succeds, but the addition fails, we can "rollback" the first subtraction to left database in same state as it was before.

#jdbc program

```
import java.sql.*;
public class StudentInfo {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/dbname";
        String user = "your_username";
        String password = "your_password";
        String sqlQuery = "SELECT name FROM student
WHERE district = 'Kathmandu'";
        try (Connection conn =
DriverManager.getConnection(url, user, password);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sqlQuery)) {
            System.out.println("Names of students living in
Kathmandu district:");
            while (rs.next()) {
                System.out.println(rs.getString("name"));
            }
        } catch (SQLException e) {
            e.printStackTrace(); } }}
```

Network Programming

The term network programming refers to writing programs that execute accross multiple devices (computers), in which the devices are all connected to each other using a network. It is also referred as socket programming because we use the concept of socket to write programs that can communicate in the network. The java.net package provides support for the two common network protocols TCP & UDP.

Transmission Control Protocol (TCP):

TCP stands for Transmission Control Protocol. It is a transport layer protocol that ease the transmission of packets from source to destination. This protocol is used with an IP protocol, so together, they are referred to as a TCP/IP.

The main functionality of the TCP is to take the data from application layer. Then it divides the data into a several packets, provides numbering to these packets, and finally transmits these packets to the destination. The TCP, on the other side, will reassamble the packets and transmits them to the application layer. As we know that TCP is a connection-oriented protocol, so the connection will remain established until the communication is not completed between the sender and the receiver.

Features of TCP Protocol:
- → Transport Layer Protocol
- → Reliable
- → Order of data is maintained.
- → Connection–oriented
- → Full duplex
- → Stream–oriented

User Datagram Protocol (UDP):

UDP is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival. UDP is not connection based like TCP.

TCP vs. UDP: [Imp]

- → TCP is a connection–oriented protocol, whereas UDP is a connectionless protocol.
- → The speed for TCP is slower while the speed of UDP is faster.
- → TCP uses handshake protocol while UDP uses no handshake protocol.
- → TCP does error checking and also makes error recovery, on the other hand, UDP performs error checking, but it discards packets having error.
- → TCP has acknowledgement segments, but UDP does not have.
- → TCP is heavy-weight, and UDP is lightweight.

When to use UDP and TCP?

- → TCP is an ideal choice, when most of the overhead is in the connection. Use TCP sockets when both client and server independently send packets. Example: Online Poker. Use TCP when occasional delay is acceptable.
- → UDP is an ideal to use with multimedia like VoIP. We should use UDP if both client and server may separately send packets. Example: Multiplayer games. Use UDP when occasional delay is not acceptable.

TCP Port:

TCP port is a unique number assigned to different applications. For example, we have opened the email and game applications on our computer; In order to do these tasks, different unique numbers are assigned to email and game. TCP and UDP protocols mainly use port numbers.

A port number is a unique identifier used with an IP address. A port is a 16-bit unsigned integer, and the total number of ports available in the TCP/IP model is 65,535 ports. Therefore the range of port numbers is 0 to 65535. In case of TCP, the zero-port number is reserved and cannot be used whereas, in UDP, the zero-port number is ~~unavailable~~ not available. IANA (Internet Assigned Numbers Authority) is a standard body that assigns the port numbers.

Example of port number:

192.168.1.100:7

In this case, 192.168.1.100 is an IP address and 7 is port number.

Why do we require port numbers?

A single client can have multiple connections with the same server or multiple servers. The client may be running multiple applications at the same time. When the client tries to access some service, then the IP address is not sufficient to access the service. To access the service from a server, the port number is required. So, the transport layer plays a major role in providing multiple communication between these applications by assigning a port number to the applications.

IP Address Network Classes in JDK:

- i) Socket Class: The Java Socket class is used to create sockets when we use TCP for communication.
- ii) Server Socket Class: It is used to create sockets for servers when TCP is used for communication.
- iii) IntAddress Class: It represents an IP address.
- iv) Datagram Socket Class: It is used to create sockets for servers when UDP is used for communication.
- v) Datagram Packet Class: This class is used to create datagram packets that are exchanged between UDP clients and UDP servers.
- vi) URL Class: It represents an URL (Uniform Resource Locator). It points to resource on World Wide Web.
- vii) URL Connection Class: It represents a communication link between the URL and the application.

What is Socket? [Imp]

A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application data that is to be sent to destination. Sockets provide communication mechanism between two programs using TCP.

Socket Programming using TCP:

How can you write java programs that communicate with each other using TCP Sockets? [Imp]

Solution:

Steps for Writing Client Program:
- → Open a socket
- → Open an input stream and output stream to the socket
- → Read from and write to the sockets stream.
- → Close the socket.

Steps for Writing Server Program:
- → Create the Object of ServerSocket class.
- → Accept the connection from the client.
- → Get input and output stream of socket.
- → Read/Write Data to the socket.
- → Close Streams and Socket.

(last three points are same to above last three points of client program only close streams added.)

Example (Client Program):

```
import java.net.*;
import java.io.*;
import java.util.*;
public class MsgClient{
    public static void main(String args[]) throws IOException{
        Socket cs = new Socket ("localhost", 1254);

        Scanner ins = new Scanner(cs.getInputStream());
        PrintWriter outs = new PrintWriter(cs.getOutputStream(), true);

        outs.println("Hello Server");
        String s = ins.nextLine();
        System.out.println("From Server: "+s);

        ins.close();
        outs.close();
        cs.close();
    }
```

(1254 is port number to which connection made)

29.

Example: (Server Program):

```
import java.net.*;
import java.io.*;
import java.util.*;
class MsgServer{
    public static void main(String args[]) throws IOException{
        ServerSocket ss = new ServerSocket(1254);

        Socket cs = ss.accept();

        Scanner ins = new Scanner(cs.getInputStream());
        PrintWriter outs = new PrintWriter (cs.getOutputStream(), true);

        String s = ins.nextLine();
        System.out.println("From Client: "+s);
        outs.println("Hello Client");

        outs.close();
        ins.close();
        cs.close();
        ss.close();
    }
}
```

(Same as Client program only these two start lines are only different and ss.close() added at last all other is same)

Socket Programming using UDP:

Steps of Writing Client Program:
→Get a datagram socket
→Send request
→Get response
→Display response
→Close Socket

Steps of Writing Server Program:
→Get a datagram socket
→Receive request
→Send response
→Close socked.

Example (Client Program):

```
import java.net.*;
import java.io.*;
public class UDPClient{

    public static void main(String args[]) throws IOException{
        DatagramSocket socket = new DatagramSocket();

        byte[] buf = new byte[256];
        InetAddress address = InetAddress.getByName("localhost");
        DatagramPacket packet = new DatagramPacket(buf, buf.length,
                address, 4445);
        socket.send(packet);

        packet = new DatagramPacket(buf, buf.length);
        socket.receive(packet);

        String received = new String(packet.getData());

        System.out.println("Quote of the Moment:" + received);
        socket.close();
    }
}
```

Example (Server Program):

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]) throws IOException{
        byte[] buf = new byte[256];

        DatagramSocket socket = new DatagramSocket(4445);

        DatagramPacket packet = new DatagramPacket(buf, buf.length);
        socket.receive(packet);
        InetAddress address = packet.getAddress();
        int port = packet.getPort();
        String s = "Hello";
        buf = s.getBytes();

        packet = new DatagramPacket(buf, buf.length, address, port);
        socket.send(packet);

        socket.close();
    }
}
```

Working with URL's:

https://www.javatpoint.com//java-tutorial
protocol — Host name — File

A URL contains following informations:
1. **Protocol:** In this case https is the protocol.
2. **IP Address or Server name:** In this case, www.javatpoint.com is the Server name.
3. **Port number:** It is an optional attribute. If is written with IP address to identify the resource uniquely. Default value is -1.
4. **File Name:** In this case, java-tutorial is file name. It can also be path to directory.

URL Demo Example:

```
import java.net.*;
public class URLDemo{
    public static void main(String[] args){
        try {
            URL url = new URL("https://www.javatpoint.com/java-tutorial");
            System.out.println("Protocol:" + url.getProtocol());
            System.out.println("Host Name:" + url.getHost());
            System.out.println("File Name:" + url.getFile());
        } catch (Exception e) {System.out.println(e);}
    }
}
```

Working with URL Connection Class:

The Java URLConnection class represents a communication link between the URL and the application. This class can be used to read and write data to the specified resource referred by the URL.

The URLConnection class provides many methods, we can display all the data of a webpage by using the getInputStream() method. The getInputStream() method returns all the data of the specified URL in the stream that can be read and displayed.

Java Mail API:

The JavaMail is an API that is used to compose, write and read electronic messages (emails). This API provides protocol-independent and platform-independent framework for sending and receiving mails. The javax.mail and javax.mail.activation packages the core classes of JavaMail API.

If we use Java 2 Platform, no additional setup is required, for Enterprise Edition (J2EE) 1.3. However, if we use Standard Edition (J2SE) 1.1.7 and upwards, we need to download and install following:
→JavaMail API
→Java Activation Framework

Sending and Receiving Email:

Steps (Sending): There are following three steps to send email using JavaMail. They are as follows:

Get the session object: Stores all the information of host like host name, username, password etc.
```
Properties properties = new Properties();
Session session = Session.getDefaultInstance(properties,null);
```
Compose the message: MimeMessage class is mostly used for composing.
```
MimeMessage message = new MimeMessage(session);
```
Send the message: Transport class provides method to send the message.
```
Transport.send(message);
```

Steps (Receiving):
→ Get the session object.
→ create the POP3 store object and connect with the pop server.
→ create the folder object and open it.
→ retrieve the messages from the folder in an array and print it.
→ close the store and folder objects.

GUI with JavaFX

JavaFX is a Java library used to develop Desktop applications as well as Rich Internet Applications (RIA). The applications built in JavaFX can run on multiple platforms including Web, Mobile and Desktops. JavaFX is intended to replace Swing in Java applications as a GUI framework. It provides more functionalities than swing. It supports various operating systems including Windows, Linux and Mac OS.

JavaFX vs. Swing:

Swing	JavaFX
1. Swing is the standard toolkit for Java developer, in creating GUI.	1. JavaFX provides platform support for creating desktop applications.
2. Swing has a more sophisticated set of GUI components.	2. JavaFX has decent but lesser number of GUI components.
3. Swing does not have support for customization using CSS and XML.	3. JavaFX has a support for customization using CSS and XML.
4. With Swing, it is very difficult to create beautiful 3-D applications.	4. With JavaFX one can also create beautiful 3-D applications.
5. Swing has a UI component library and act as a legacy.	5. JavaFX has several components built over Swing.

JavaFX Layouts:

JavaFX has following built-in layout panes: FlowPane, HBox, VBox, BorderPane, GridPane.

FlowPane: FlowPane layout organizes the nodes in a flow that are wrapped at the flowpane's boundry. The horizontal flowpane arranges the nodes in a row and wrap them according to the flowpane's width. The vertical flowpane arranges the nodes in a column and wrap them according to the flowpane's height.

alignment, columnHalignment, hgap, orientation, vgap, rowValignment etc. are various properties of FlowPane class. FlowPane(), FlowPane(Double Hgap, Double Vgap), FlowPane(Node... children), FlowPane(Orientation orientation) etc. are some constructors of FlowPane.

BorderPane: BorderPane arranges the nodes at the left, right, centre, top, and bottom of the screen. This class provides various methods like setRight(), setLeft(), setCenter(), setBottom(), and setTop() which are used to the set the position for the specified nodes.

Bottom, Centre, left, Right, and Top are the properties of BorderPane class. BorderPane(), BorderPane(Node Center), BorderPane(Node Center, Node top, Node right, Node bottom, Node left) are its constructors.

GridPane: GridPane layout allows us to add the multiple nodes in multiple rows and columns. It is seen as a flexible grid of rows and columns where nodes can be placed in any cell of the grid.

alignment, gridLinesVisible, hgap, vgap are its properties. This class contains only one constructor, Public GridPane(), which creates a gridpane with 0 hgap/vgap.

HBox: HBox layout pane arranges the nodes in a single row. alignment, fillHeight, spacing are its properties. HBox(), HBox(Double spacing) are its constructors.

VBox: VBox layout pane arranges the nodes in a single column. alignment, fillWidth, Spacing are its properties. VBox(), VBox(Double Spacing), VBox(Double spacing, Node? children), VBox(Node? children) are its constructors.

JavaFX UI Controls:

The UI elements are the one which are actually shown to the user for interaction or information exchange. Layout defines the organization of the UI elements on the screen. Behaviour is the reaction of the UI element when some event is occured on it.

The package javafx.scene.control provides all the necessary classes for the UI components like Button, Label etc. Every class represents a specific UI control and defines some methods for their styling.

Label: Label is a component that is used to define a simple text on the screen. Typically, a label is placed with the node, it describes. javafx.scene.control.Label class represents label control. Label(), Label(String text), Label (String text, Node graphics), are its constructors.

TextField: Text Field is basically used to get input from the user in the form of text. javafx.scene.control.TextField represents TextField. It provides various methods to deal with textfields in JavaFx.

Button: Button is a component that controls the function of the application. Button class is used to create a labelled button. javafx.scene.control.Button class represents Button. An event is generated whenever the button gets clicked.

RadioButton: The Radio Button is used to provide various options to the user. The user can only choose one option among all. A radio button is either selected or deselected. It can be used in a scenario of multiple choice questions like in quiz where one option needs to be choosen.

CheckBox: Check Box is used to get the kind of information from the user which contains various choices. User marks the checkbox either on (true) or off (false). It can be used in a scenario where the user is prompted to select more than one option.

Hyperlink: Hyperlink are used to refer any of the webpage through our application. It is represented by the class javafx.scene.control.Hyperlink. It is similar to anchor links in HTML.

Menu: JavaFx provides a Menu class to implement menus. Menu is the main component of any application. javafx.scene.control.Menu class provides all the methods to deal with menus.

ToolTip: JavaFx ToolTip is used to provide hint to the user about any component. It is mainly used to provide hints about the textfields or password fields being used in the application.

FileChooser: JavaFx FileChooser enables users to browse the files from the file system. The FileChooser class provides two types of methods: showOpenDialog() and showSaveDialog(). javafx.stage.FileChooser class represents FileChooser.

Steps of creating GUI using JavaFx:

Step1: Extend javafx.application.Application and override start().

Step2: Create any UI control component using JavaFx GUI, according to our need.

Step3: Create any Layout and add UI component to it.

Step4: Create a Scene by instantiating javafx.scene.Scene class.

Step5: Prepare the stage with javafx.stage.Stage class.

Step6: Create an event for the UI component if any and call setOnAction() method for handling event, in which we can define any method which contains code for how event is handeled.

Step7: Create the main method to launch the application.

Servlets and Java Server Pages

Web Container:

Web container is runtime environment for a web application. The web applications run within web container. Web container may be built-in with web server or sometimes it can be seperate system. When a request is made from client to web server, it passes the request to web container, which finds the correct resource to handle the request, and then use the response from the resource to generate response and provide it to web server. Some of the important works done by web container are:
→Communication support between web server and servlets and JSPs.
→Lifecycle and Resource Management of servlet.
→Multithreading support.
→JSP Support
→Miscellaneous Task.

Introduction to Servlet:

Servlets are small programs that execute on the server side of a Web connection. Just as applets dynamically extend the functionality of a web server, servlets dynamically extend the functionality of a Web server. It is a Java programming language class used to extend the capabilities of servers that host applications accessed via a request-response programming model. Although servlets can respond to any type of requests, they are commonly used to extend the applications hosted by Web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes. The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.

Life Cycle of a Servlet:

The Servlet life cycle has 5 stages in general. Among them three methods are central to the life cycle of a servlet. These are init(), service(), and destroy().

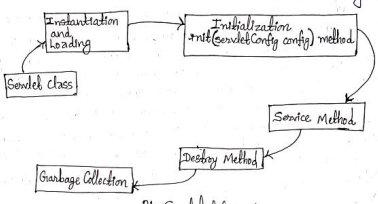


Fig: Servlet life cycle.

1. **Loaded and Instantiated:** The class loader is responsible to load the servlet class. Then it creates the instance of a servlet after loading the servlet class. The servlet engine can instantiate more than one servlet instance. The servlet instance is created only once in the servlet life cycle.
2. **Initialized by calling the init() method:** The init method is used to initialize the servlet. Syntax of the init method is as follows:
   `public void init(ServletConfig config) throws ServletException`
3. **Process a client's request by invoking service() method:** The web container calls the service method each time when request for the servlet is received. The syntax is as follows:
   `public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException`
4. **Terminated by calling the destroy() method:** It gives the servlet an opportunity to clean up any resource for example memory, thread etc. Syntax: `public void destroy()`
5. **Garbage collected by the JVM:** Once the servlet is destroyed, garbage collector component of JVM is responsible collecting the garbage's.

Servlet APIs:

Two packages contain the classes and interfaces that are required to build servlets. These are javax.servlet and javax.servlet.http. They constitute the Servlet API. These packages are not part of the Java core packages. Instead, they are standard extensions. Therefore, they are not included in the Java Software Development Kit. We must download Tomcat or Glassy Fish server to obtain their functionality.

The javax.servlet Package: This package contains a number of interfaces and classes that establish the framework in which servlets operate. Following are some of the interfaces and classes:

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

Class	Description
GenericServlet	Implements the Servlet and ServletConfig interfaces.
ServletInputStream	Provides an input stream for reading requests.
ServletOutputStream	Provides an output stream for writing responses.
ServletException	Indicates a servlet error occurred.

The javax.servlet.http Package: This package contains a number of interfaces and classes that are commonly used by servlet developers. Following are some of the interfaces and classes:

Interface	Description
HttpServletRequest	Enables servlets to read data from HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.

Class	Description
Cookie	Allows state information to be stored on client machine
HttpServlet	Provides methods to handle HTTP requests and responses.
HttpSessionEvent	Encapsulates a session-changed event.

⊛ Writing Servlet Programs:

The servlet program can be created by three ways:
i) By implementing servlet interface
ii) By inheriting GenericServlet class
iii) By inheriting HttpServlet class.

The mostly used approach is by extending HttpServlet because it provides http request specific method such as doGet(), doPost(), doHead() etc.

There are six steps to write Servlet Program (Here we are using apache tomcat server):
i) Create a directory structure
ii) Create a Servlet
iii) Compile the Servlet
iv) Create a deployment descriptor
v) Start the server and deploy the project
vi) Access the servlet

Example: DemoServlet.java

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
public class DemoServlet extends HttpServlet{
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException{
    res.setContentType("text/html"); //setting the content type.
    PrintWriter pw=res.getWriter(); //get the stream to write data.
    //writing html in the stream
        pw.println("<html> <body>");
        pw.println("Welcome to servlet");
        pw.println("</body></html>");

    pw.close(); //closing the stream
??
```

⊛ Processing Forms using Servlet: [Imp]

HTML Forms collect data from the user via input elements and then:
→ Request will be sent from HTML Form
→ Request will be either GET/POST method in servlet side
→ Response will be provided from servlet as an HTML form.

Usually sensitive data such as password should be sent using POST method, because GET method passes information from browser to web server producing a long string in browser's URL which can be accessible.

Servlet reads parameters in a form using the following methods depending on the situation:
- getParameter(): We call request.getParameter() method to get the value of a form parameter.
- getParameterValues(): We call this method if the parameter appears more than once, and returns multiple values, for example checkbox.
- getParameterNames(): We call this method if we want a complete list of all parameters in the current request.

Example: Simple Servlet program that reads and displays data from HTML Form. (using POST method).

(Get Method भने doPost को सट्टा doGet लेख्ने अरु same. Question मा: How do you handle HTTP request (POST/GET) using Servlet भनेर पनि यदि example लेख्दा हुन्छ।)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class FormServlet extends HttpServlet{
    public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    //Get the request paramaters
    String username= request.getParameter("username");
    String password= request.getParameter("password");
    //set the response content type
    response.setContentType("text/html");
    //Get the output stream for writing the response.
    PrintWriter out= response.getWriter();

//Print the response
    out.println("<html>");
    out.println("<head>");
    out.println("<title> Form Servlet </title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1> Form Servlet </h1>");
    out.println("<p> Username: " +username + "</p>");
    out.println("<p> Password: " +password + "</p>");
    out.println("</body>");
    out.println("</html>");
  }
}
```

(→ reading using getParameter())

⊛ Database Access with Servlets:

(Almost many things are exactly similar to database connection in unit-4)

To access a database from a servlet, we need to do following:
1) Load the database driver class.
```
Class.forName("com.mysql.jdbc.Driver");
```
2) Establish a connection to the database:
```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/mydatabase", "username", "password");
```
3) Create a statement object:
```
Statement stmt = con.createStatement();
```
4) Execute a SQL Statement.
```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
```
5) Process the result set using various methods of 'ResultSet' object such as 'next', 'getString', 'getInt' etc.
6) Finally close the connection.

⊛ Handling Cookies in Servlet:

(→ Short concepts of cookies and session [Imp])

Cookies and sessions are two common mechanisms used in web application development to store information about a user and track their activity across multiple requests.

Cookies are small pieces of data that are stored by the browser on the clients computer. They can be used to store information such as user's preferences or login status. In servlet we can use the 'javax.servlet.http.Cookie' class to create, read and delete cookies.

⇒ To create a cookie and add it to the response, we can use the 'addCookie' method of the 'HttpServletResponse' object. For Example:
```
Cookie cookie= new Cookie("username", "john");
response.addCookie(cookie);
```
⇒ To read a cookie from the request, we can use the 'getCookies' method of the 'HttpServletRequest' object. This returns an array of Cookie objects, which we can loop through to find the one we are looking for. For Example:
```
Cookie[] cookies= request.getCookies();
if (cookies != null){
  for(Cookie cookie: cookies){
    if(Cookie.getName().equals("username")){
      String username= cookie.getValue();
        //do something with username like print
    }
  }
}
```
⇒ To delete a cookie, we can set its maximum age to 0 and add it to the response. For example:
```
Cookie cookie= new Cookie("username", " ");
cookie.setMaxAge(0);
response.addCookie(cookie);
```

⊛ Handling Sessions in Servlet:

Sessions are used to store information about a user across multiple requests. When a user visits a web application, the server creates a unique session ID and sends it back to the browser in the form of a cookie.

In a servlet, we can use the 'getSession' method of the 'HttpServletRequest' object to access the current session. This method returns a 'javax.servlet.http.HttpSession' object, which we can use to store and retrive information about the user. To store an attribute in the session, we can use the 'setAttribute' method of the 'HttpSession' object. For Example:
```
HttpSession session= request.getSession();
session.setAttribute("username", "roshan");
```
⇒ To retrive an attribute from the session, we can use the 'getAttribute' method of the 'HttpSession' object. For example:
```
HttpSession session= request.getSession();
String username= (String) session.getAttribute("username");
```
⇒ To invalidate the current session and remove all attributes, we can use the 'invalidate' method of the 'HttpSession' object. For example:
```
HttpSession session= request.getSession();
session.invalidate();
```

⊛ Servlet vs. JSP: [Imp]

Servlets	JSP
i) Servlet is a pure java code	i) JSP is a tag based approach.
ii) We write HTML in servlet code.	ii) We write JSP code in HTML.
iii) Servlet is faster than JSP	iii) JSP is slower than Servlet.
iv) Writing code for servlet is harder.	iv) Writing code for JSP is easier.
v) Servlet can accept all protocol requests.	v) JSP only accept http requests.
vi) Modification in Servlet is a time consuming task because it includes reloading.	vi) JSP modification is fast, we just need to click refresh button.
vii) Servlet do not have implict objects	vii) JSP have implicit objects.
viii) In MVC pattern, servlet act as a controller.	viii) In MVC pattern, JSP act as a view.

⊛ JSP Access Model:

In JSP, the access model refers to the way in which the built-in objects and other resources are made available to the JSP page. There are three main access models in JSP, which are used ~~use~~ according to the need of application:

i) **Page-Based Access:** In this model, the built-in objects are made available to the JSP page through local variables.
Syntax: 'request.getParameter("paraName")';

ii) **Action-Based Access:** In this model, the built-in objects are made available to the JSP page through the use of custom tags.

iii) **Bean-Based Access:** In this model, the built-in objects are made available to the JSP page through JavaBeans.
Syntax: 'beanName.propertyName'.

Q. What is JSP? Discuss with suitable example. [Imp]

Java Server Pages (JSP) is a server side technology to create dynamic java web application. It allows java programming code to be embedded in the HTML pages. JSP provides following scripting elements (i.e, JSP Syntaxes):

JSP Comments: Syntax: <%-- comments --%>

JSP Scriplet: This tag is used to execute java code in JSP.
Syntax: <% java code %>

JSP Expression: It is used to insert a single java expression directly into the response message, by placing inside a out.print() method.
Syntax: <%= Java Expression %>

JSP Declaration: This tag is used to declare variables and methods.
Syntax: <%! Variable or Method declaration %>

Example: Let we take a simple JSP program to display "Tribhuwan University" 10 times, as our example.

Solution:
```
<!DOCTYPE html>
<html>
    <head> <title>Sample JSP Program </title>
    </head>
    <body> <h1>Displaying "Tribhuwan University" 10 times.</h1>
  <table>
      <%
        for (int i=1; i<=10; i++){
      %>
        <tr><td>Tribhuwan University</td> </tr>
      <% } %>
  </table>
</body>
</html>
```

(Here we use ... to display to ... used. We can also print directly as Tribhuwan University without any tags)

⊛ JSP Implicit Objects:

In JSP, implicit objects are objects that are automatically available to the JSP page and do not need to be explicitly created. Some of the implicit objects in JSP are as follows:

i) **request:** It Represents the HTTP request received by the web server and provides access to request-specific information such as parameters, headers, and session attributes.
ii) **response:** It Represents the HTTP response that will be sent back to the client and allows the JSP to control the content and headers of the response.
iii) **pageContext:** Provides access to various components of the JSP environment, such as the ServletContext, HttpSession, and JspWriter.
iv) **session:** Represents the HTTP session for a user and allows the JSP to access and modify session-level data.
v) **page:** Represents the current JSP page and provides access to the page's attributes and methods.

These implicit objects can be accessed using standard Java syntax within a JSP page. For example, to access the request object, we would use 'request', and to access a parameter passed in the request, we would use 'request.getParameter("paramName

⊛ JSP Syntax for Directions:

(→ JSP Syntax मा direction हुदैन so यता लेखेको)

Syntax: <%@ directive %>

It is used to specify various attributes for the page or to include other resources in the page.

Examples:
```
<%@ attribute="value" %>
<%@ include file="filename" %>
```

⊛ Object Scope: [Imp]

In JSP, object scope refers to the availability of an object within the page, application, or session. The scope of an object determines where it can be accessed and for how long it is available. There are four object scopes in JSP:

i) **Page scope:** An object with page scope is available only within the current JSP page and is not accessible from other pages.
ii) **Request scope:** An object with request scope is available within the current request and response cycle and is ~~is~~ not accessible from other requests.
iii) **Session scope:** An object with session scope is available to all pages within a user's session and is not accessible from other sessions.
iv) **Application scope:** An object with application scope is available to all pages in the web application and is not accessible from other applications.

⊛ Processing Forms in JSP: [Imp]

To process a form in a JSP page, we can use a combination of HTML and JSP to create the form and handle the form submission. Here is an example of simple form that allows user to enter their name and submit the form:
```
<form action="processForm.jsp" method="post">
    Name: <input type="text" name="name"> <br>
        <input type="submit" value="Submit">
</form>
```
To process the form submission in the JSP page, we can use the 'request' implicit object to access the form data. For example:
```
<%
    String name= request.getParameter("name");
    //process the form data
%>
```
This code uses the 'getParameter' method of the 'request' object to retrive the value of the "name" parameter from the form submission. We can then use this value to process the form data as needed.

⊛ Database Access with JSP:

To access a database from a JSP page, we will need to follow following steps:
→ Configure a JDBC driver and connection pool.
→ Establish a database connection.
→ Execute a SQL query
→ Process the result set
→ Close the connection.

Example:
```
<%@page import="java.sql.*"%>
<%
  //Load the JDBC driver
  Class.forName("com.mysql.jdbc.Driver");
 //Configure the connection pool
  String url="jdbc:mysql://localhost:3306/mydatabase";
  String username ="myuser";
  String password="myuser123";
  Connection pool=DriverManager.getconnection(url, username, password);
//Establish a connection
  Connection conn=pool.getConnection();
//Execute a SQL query
  Statement stmt=conn.createStatement();
  ResultSet rs=stmt.executeQuery("SELECT * FROM mytable");
//Process the result set
  while (rs.next()){
      String name=rs.getString("name");
      out.println("Name: "+name + "<br>");
  }
//Close the connection
  conn.close();
%>
```
This example loads the MySQL JDBC driver, establishes a connection to a database named "mydatabase", executes a SQL query to select all rows from a table named "mytable", and prints the "name" column of each row to web page.

⊛ Introduction to Java Web Frameworks:

A web framework is a collection of libraries and tools that make it easier to develop web applications. Web Frameworks provide a standard way to build and deploy web applications, and they can help to save time and effort. There are many Java web frameworks available, and each has its own strengths and limitations. Some popular Java web frameworks include:

i) **Spring:** Spring is widely used framework for building Java applications. It provides a range of features including dependency injection, data access, and web support.
ii) **Hibernate:** Hibernate is an object-relational mapping (ORM) framework that simplifies the process of storing and retriving data from a database. It is often used with Spring.
iii) **Struts:** Struts is an older web framework that is still used by some developers. It uses a MVC architecture and provides support for actions, forms, and validation.
iv) **JSF:** It is a framework for building user interfaces for Java Web applications. It provides suppart for actions, forms, validation. JSF stands for JavaServer Faces.
v) **Play:** Play is a modern web framework that is designed to be easy to use and scalable. It uses an event-driven architecture and supports real-time web applications.

⊛ Introduction of RMI:

(→ RMI def^n [Imp])

The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. A distributed system, also known as distributed computing, is a system with multiple components located on different machines that communicate and coordinate actions in order to appear as a single coherent system to the end-user. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects stub and skeleton.

Goals of RMI:
→ To minimize the complexity of the application.
→ To preserve type safety.
→ Distributed garbage collection.
→ Minimize the difference between working with local and remote objects.

Requirements of distributed applications:
If any application performs these tasks, it can be distributed application.
→ The application need to locate the remote method.
→ It need to provide the communication with the remote objects, and
→ The application need to load the class definitions for the objects

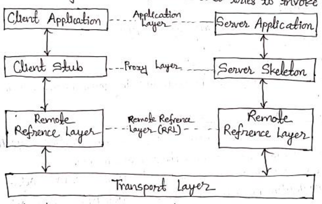
Understanding stub and skeleton:

Stub: The stub is an object, that acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:
→It initiates a connection with remote Virtual Machine (JVM)
→It writes and transmits the parameters to remote Virtual Machine.
→It waits for the result.
→It reads the return value.
→ Finally returns the value to the caller.

Skeleton: The skeleton is an object, that acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:
→It reads the parameter for the remote method.
→It invokes the method on the actual remote object.
→It writes and transmits the result to the caller.

⊛Architecture of RMI: [Imp]

In an RMI application, we write two programs, a server program and a client program. Inside the server program, a remote object is created and refrence of that object is made available for the client. The client program requests the remote objects on the server and tries to invoke its methods.



Application layer: In this layer client and server are involved in communication. The java program on client side communicates with the java program on server side.

Proxy layer: This layer contains the client stub and server skeleton objects.
Stub: A stub is an object that acts as a gateway for client side. All the outgoing requests are routed through it.
Skeleton: The skeleton is an object, that acts as a gateway for server side. All the incomming requests are routed through it.

Remote Refrence Layer (RRL): This layer is responsible to maintain session during the method call. It is also responsible for handling duplicated objects.

Transport Layer: It is responsible for setting up communication between two machines. This layer uses standard TCP/IP protocol for connection.

⊛Creating and Executing RMI Applications: [Imp] (Question यसरी सोध्दा सक्छ)

Q. How can you use RMI to develop a program that runs in different machine? Discuss with suitable example.

OR Q. Describe the process to run the RMI application.

Solution:

To create an RMI application in Java, we will need to do the following steps:
→Define the remote interface
→Implement the remote interface.
→Create the server.
→ Create the client.

Example:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
//Define the remote interface
public interface Hello extends Remote{
    String sayHello() throws RemoteException;
}
//Implement the remote interface
public class HelloImpl extends UnicastRemoteObject implements Hello{
    public HelloImpl() throws RemoteException{
            super();
    }
    public String sayHello(){
        return "Hello, World!";
    }
}
//Create the server
public class Server{
    public static void main(String[] args){
        try{ HelloImpl obj=new HelloImpl(); //Create remote object
            //Blind the remote object to the RMI registry
            Naming.rebind("//localhost/Hello", obj);
            System.out.println("Hello Server ready.");
        }catch (Exception e){
    System.out.println("Server failed:" +e);
    }
  }
}
//Create the client
public class Client{
    public static void main (String[] args){
        try{
            //Lookup the remote object
            Hello obj= (Hello) Naming.lookup("//localhost/Hello");
            //Invoke a method on the remote object
            String message=obj.sayHello();
            System.out.println(message);
        }catch (Exception e){
            System.out.println("Hello Client exception: "+e);
        }
    }
}
```

To compile and run, we need to include the 'rmi.jar' file in our classpath and start the RMI registry with 'rmiregistry' command. Then, we can compile the server and client classes using 'javac' and run the client and server using 'java' as follows:

```
javac Server.java HelloImpl.java
java Server

javac Client.java
java Client
```

This will cause the client to connect.

⊛Introduction to CORBA: [Imp]

The Common Object Request Broker Architecture (CORBA) is a standard developed by the Object Management Group (OMG) to provide interoperability among distributed objects. CORBA is the world's leading middleware solution enabling the exchange of information, independent of hardware platforms, programming languages and operating systems. CORBA is often described as a "software bus" because it is a software-based communications interface through which objects are located and accessed.

RMI	CORBA
i) RMI is a Java-specific technology.	i) CORBA is independent of programming languages.
ii) It uses Java interface for implementation.	ii) It uses Interface Definition Language (IDL) to seperate interface from implementation.
iii) RMI programs can download new classes from remote JVMs.	iii) CORBA doesn't have this code sharing mechanism.
iv) RMI passes objects by remote refrence or by value.	iv) CORBA passes objects by refrence.
v) Distributed garbage collection is available.	v) Distributed garbage collection is not available.
vi) Generally simpler to use	vi) More complicated.
vii) It is free of cost	vii) Cost money according to the vendor.

⊛CORBA Architecture:

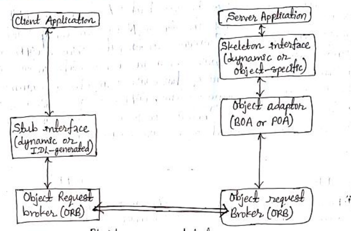


Fig: The CORBA Architecture

The CORBA architecture consists of following components:

Object Request Broker (ORB): The ORB is the core component of the CORBA architecture. It is responsible for handling communication between objects and for locating and activating remote objects.

IDL compiler: The IDL compiler is used to generate code from IDL interfaces. It generates code for the client-side and server-side stubs and skeletons, which are used to communicate with the remote object.

Client: The client is a program that invokes operations on a remote object. It uses the client-side stub to communicates with the ORB, which in turn communicates with server-side skeleton.

Server: The server is a program that implements the operations of a remote object. It uses the server-side skeleton to communicate with the ORB, which in turn communicates with the client-side stub.

⊛Interface Definition Language (IDL): [Imp]

The Interface Definition Language (IDL) is a language that is used to define interfaces for distributed objects in the CORBA. IDL is a language-neutral way of specifying the interface to a remote object, and it allows software components written in different programming languages to work together.

In IDL an interface is defined as a set of operations that can be invoked on a remote object. Each operation has a name, return type, and a list of parameters. Here is an example of an IDL interface that defines a single operation, 'sayHello', which takes no parameters and returns a string:

```
interface Hello{
    string sayHello();
}
```

⊛Simple CORBA Program:

```
module HelloApp
{
    interface Hello
    {
        string sayHello("Hello");
    };
};
```

#rmi to find product
-define interface

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface ProductInterface extends Remote {
    int getProduct(int num1, int num2) throws
RemoteException;
}
```

-implement interface

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class ProductImpl extends UnicastRemoteObject
implements ProductInterface {
    protected ProductImpl() throws RemoteException {
        super();
    }
    public int getProduct(int num1, int num2) throws
RemoteException {
        return num1 * num2;
    }}
```

-rmi server

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class ProductServer {
    public static void main(String[] args) {
        try {
            ProductInterface productImpl = new ProductImpl();
            Registry registry =
LocateRegistry.createRegistry(1099);
            registry.bind("ProductService", productImpl);
            System.out.println("Server is running...");
        } catch (Exception e) {
            System.err.println("Server exception: " +
e.toString());
            e.printStackTrace();
        }}}
```

-client

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ProductClient {
    public static void main(String[] args) {
        try {
            Registry registry =
LocateRegistry.getRegistry("localhost");
            ProductInterface product = (ProductInterface)
registry.lookup("ProductService");
            int num1 = 5;
            int num2 = 10;
            int result = product.getProduct(num1, num2);
            System.out.println("Product of " + num1 + " and "
+ num2 + " is: " + result);
        } catch (Exception e) {
            System.err.println("Client exception: " +
e.toString());
            e.printStackTrace();}}}
```

#login form with layout manager

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class LoginForm {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Login Form");
        frame.setSize(300, 200);
        JPanel panel = new JPanel(new GridLayout(4, 2, 5, 5));
         JLabel lblUserId = new JLabel("User ID:");
        JTextField txtUserId = new JTextField(10);
        JLabel lblPassword = new JLabel("Password:");
        JPasswordField txtPassword = new JPasswordField(10);
        JLabel lblAccountType = new JLabel("Account Type:");
        String[] accountTypes = {"admin", "user"};
        JComboBox<String> cmbAccountType = new
JComboBox<>(accountTypes);
        JButton btnOk = new JButton("OK");
        btnOk.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                String userId = txtUserId.getText();
                String password = new
String(txtPassword.getPassword());
                String accountType = (String)
cmbAccountType.getSelectedItem();
                System.out.println("User ID: " + userId);
                System.out.println("Password: " + password);
                System.out.println("Account Type: " + accountType);
            }});
        panel.add(lblUserId);
        panel.add(txtUserId);
        panel.add(lblPassword);
        panel.add(txtPassword);
        panel.add(lblAccountType);
        panel.add(cmbAccountType);
        panel.add(new JLabel()); // Empty cell for spacing
        panel.add(btnOk);
        frame.add(panel);
        frame.setVisible(true);
    }}
```

#login form using JavaFX

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
public class LoginForm extends Application {
    public static void main(String[] args) {
        launch(args); }
    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Login Form");
        GridPane grid = new GridPane();
        grid.setAlignment(javafx.geometry.Pos.CENTER);
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(25, 25, 25, 25));
        Label lblUserId = new Label("User ID:");
        grid.add(lblUserId, 0, 0);
        TextField txtUserId = new TextField();
        grid.add(txtUserId, 1, 0);
        Label lblPassword = new Label("Password:");
        grid.add(lblPassword, 0, 1);
        PasswordField txtPassword = new PasswordField();
        grid.add(txtPassword, 1, 1);
        Label lblAccountType = new Label("Account Type:");
        grid.add(lblAccountType, 0, 2);
        ChoiceBox<String> cmbAccountType = new
ChoiceBox<>();
        cmbAccountType.getItems().addAll("admin", "user");
        cmbAccountType.setValue("user");
        grid.add(cmbAccountType, 1, 2);
        Button btnOk = new Button("OK");
        btnOk.setOnAction(e -> {
            String userId = txtUserId.getText();
            String password = txtPassword.getText();
            String accountType = cmbAccountType.getValue();
            System.out.println("User ID: " + userId);
            System.out.println("Password: " + password);
            System.out.println("Account Type: " + accountType);
        });
        grid.add(btnOk, 1, 3);
        Scene scene = new Scene(grid, 300, 200);
        primaryStage.setScene(scene);
        primaryStage.show(); }}
```

#servlet factorial

```
    import java.io.*;import javax.servlet.*;import
javax.servlet.http.*;
    public class FactorialServlet extends HttpServlet {
        @Override
        public void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<html><head><title>Factorial
Calculator</title></head><body>");
            out.println("<h2>Factorial Calculator</h2>");
            out.println("<form method=\"post\">");
            out.println("Enter a number: <input type=\"text\"
name=\"number\"><br>");
            out.println("<input type=\"submit\" value=\"Calculate
Factorial\">");
            out.println("</form>");
            out.println("</body></html>"); }
        @Override
        public void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            String numberStr = request.getParameter("number");
            if (numberStr != null && !numberStr.isEmpty()) {
                try {int number = Integer.parseInt(numberStr);
                    long fact= calculateFactorial(number);
                    out.println("<html><head><title>Factorial
Result</title></head><body>");
                    out.println("<h2>Factorial Result</h2>");
                    out.println("Factorial of " + number + " is: " + fact);
                    out.println("</body></html>");
                } catch (NumberFormatException e)
{out.println("<html><head><title>Error</title></head><bod
y>"); out.println("<h2>Error</h2>");
                    out.println("Invalid number format.");
                    out.println("</body></html>");}} else
{out.println("<html><head><title>Error</title></head><bod
y>"); out.println("<h2>Error</h2>");
                out.println("Number field is empty.");
                out.println("</body></html>");}}
        private long calculateFactorial(int n) { if (n == 0)return 1;
            else
                return n * calculateFactorial(n - 1);
        }}
```