UNIT IV

4 MARKS

1.Name the package used for Swing components in Java?

The package used for Swing components in Java is ‘javax.swing’.

To import it, you can use the following statement:

```
import javax.swing.*;
```

2.recall the method used to create a jframe in

Java swing?

To create a JFrame in Java Swing, you typically use the ‘JFrame’ class constructor.

Here's the basic syntax:

```
JFrame frame = new JFrame("Title");
```

This creates a new JFrame with the specified title. You can further customize the JFrame by setting its size, layout, adding components, and so on.

3.Explain the concept of I/O streams in Java and their role in handling input and output operations?

Java brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.

Before exploring various input and output streams lets look at 3 standard or default streams that Java has to provide which are also most common in use:

System.in: This is the standard input stream that is used to read characters from the keyboard or any other standard input device.

System.out: This is the standard output stream that is used to produce the result of a program on an output device like the computer screen.

System.err: This is the standard error stream that is used to output all the error data that a program might throw, on a computer screen or any standard output device. This stream also uses all the 3 above-mentioned functions to output the error data:

print()

println()

printf()

4.Explain the differences between buffered streams and unbuffered streams in Java I/O?

Feature	**Buffered Streams**	**Unbuffered Streams**
Buffering	Buffered streams use an internal buffer to improve performance. They read/write data in chunks from/to the underlying stream.	Unbuffered streams do not use an internal buffer. They read/write data directly to/from the underlying stream, one byte or character at a time.
Performance	Buffered streams are generally faster for I/O operations involving larger amounts of data, as they reduce the number of actual I/O operations by reading/writing data in larger chunks.	Unbuffered streams can be slower for I/O operations involving larger amounts of data, as they read/write data byte-by-byte or character-by-character, resulting in more I/O operations.
Use Cases	Buffered streams are suitable for reading/writing data from/to files, network sockets, or other I/O sources where performance is a concern. Unbuffered streams are suitable for reading/writing data from/to sources where performance is less critical, or when	Unbuffered streams are suitable for reading/writing data from/to sources where performance is less critical, or when fine-grained control over I/O operations is required.
Flushing	Buffered streams automatically flush the buffer when the buffer is full or when explicitly requested by calling the flush() method.	Unbuffered streams do not have a buffer to flush, so there is no concept of flushing in unbuffered streams.
Constructor Parameters	Buffered streams typically wrap an	Unbuffered streams do not require

	existing unbuffered stream and require the underlying unbuffered stream to be passed as a parameter to their constructor.	any additional parameters, as they do not have a buffer
Examples	BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter	FileInputStream, FileOutputStream, FileReader, FileWriter

5.Write a Java program that demonstrates the usage of FileInputStream and FileOutputStream for file I/O operations?

Fileinputstream :

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
public class FileInputStreamExample {
  public static void main(String args[]) throws IOException {
    //Creating a File object
    File file = new File("D:/images/javafx.jpg");
    //Creating a FileInputStream object
    FileInputStream inputStream = new FileInputStream(file);
    //Creating a byte array
    byte bytes[] = new byte[(int) file.length()];
    //Reading data into the byte array
    int numOfBytes = inputStream.read(bytes);
    System.out.println("Data copied successfully...");
  }
}
```

Output

```
Data copied successfully...
```

Fileoutputstream:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class FileInputStreamExample {
  public static void main(String args[]) throws IOException {
    //Creating a File object
    File file = new File("D:/images/javafx.jpg");
    //Creating a FileInputStream object
    FileInputStream inputStream = new FileInputStream(file);
    //Creating a byte array
    byte bytes[] = new byte[(int) file.length()];
    //Reading data into the byte array
    int numOfBytes = inputStream.read(bytes);
    System.out.println("Data copied successfully...");
    //Creating a FileInputStream object
    FileOutputStream outputStream = new FileOutputStream("D:/images/output.jpg");
    //Writing the contents of the Output Stream to a file
    outputStream.write(bytes);
    System.out.println("Data written successfully...");
  }
}
```

Output

```
Data copied successfully...
Data written successfully...
```

6.Write a Java program that demonstrates the creation and execution of multiple threads?

```
 // Java code for thread creation by extending
// the Thread class

class MultithreadingDemo extends Thread {

    public void run()

    {

        try {

            // Displaying the thread that is running

            System.out.println(

                "Thread " + Thread.currentThread().getId()

                + " is running");

        }

        catch (Exception e) {

            // Throwing an exception

            System.out.println("Exception is caught");
        }

    }
}

// Main Class

public class Multithread {

    public static void main(String[] args)

    {

        int n = 8; // Number of threads

        for (int i = 0; i < n; i++) {

            MultithreadingDemo object

                = new MultithreadingDemo();

            object.start();

        }

    }
}
```

Output

```
Thread 15 is running
Thread 14 is running

Thread 16 is running

Thread 12 is running

Thread 11 is running

Thread 13 is running

Thread 18 is running

Thread 17 is running
```

7.Create a Java method that efficiently uses wait(), notify(), and notifyAll() for inter-thread communication.

1) wait() method:

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method:

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
public final void notify()
```

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

```
public final void notifyAll()
```

8.Given a complex program, analyze and illustrate how thread priorities and state transitions affect the execution flow in Java.

In Java, thread priorities and state transitions play a crucial role in determining the execution flow of a program. Let's break down how they impact the execution:

1. Thread Priorities:

Threads in Java can have priorities ranging from 1 to 10, where 1 is the lowest priority and 10 is the highest. The default priority is 5. The priority affects how the operating system schedules threads for execution.

- Higher priority threads are given preference by the scheduler, but this is not guaranteed. The actual behavior can vary between different JVM implementations and operating systems.
- Thread priorities are used to influence the order in which threads are executed, but they are not a strict guarantee of execution order.

2. Thread States:

Threads in Java can be in one of several states:

- New: The thread has been created but has not yet started.
- Runnable: The thread is ready to run and waiting for the scheduler to allocate CPU time.
- Blocked/Waiting: The thread is waiting for a monitor lock or for some other condition to be satisfied before it can proceed.
- Timed Waiting: The thread is waiting for a specified amount of time.
- Terminated: The thread has finished executing.

Now, let's illustrate how thread priorities and state transitions affect the execution flow with an example:

```java
public class PriorityExample {
  public static void main(String[] args) {
    Thread highPriorityThread = new Thread(new Worker(), "High Priority Thread");
    Thread lowPriorityThread = new Thread(new Worker(), "Low Priority Thread");

    highPriorityThread.setPriority(Thread.MAX_PRIORITY);
    lowPriorityThread.setPriority(Thread.MIN_PRIORITY);

    highPriorityThread.start();
    lowPriorityThread.start();
  }

  static class Worker implements Runnable {
    public void run() {
      System.out.println(Thread.currentThread().getName() + " is running");
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

In this example, we create two threads, one with high priority and one with low priority. The high priority thread is more likely to be scheduled first by the operating system, but it's not guaranteed.

The actual behavior depends on the JVM and the underlying operating system's scheduler implementation.

Overall, while thread priorities can influence the order in which threads are executed, they should not be relied upon for critical application logic. It's important to design your program in a way that is robust and does not depend heavily on thread priorities for correctness.

9.Given a complex program, analyze and illustrate how different network protocols in Java handle data transmission and reception?

In Java, network communication is typically handled using the `java.net` package, which provides classes for networking functionality. Let's discuss how different network protocols, such as TCP and UDP, handle data transmission and reception in Java:

1. TCP (Transmission Control Protocol)

- TCP provides a reliable, connection-oriented communication between two devices.
- In Java, TCP communication is typically implemented using the `Socket` and `ServerSocket` classes.
- Data transmission:
  - To send data over TCP, a `Socket` is created with the destination IP address and port number.
  - The data is then sent over the socket's output stream using methods like `OutputStream.write(byte[])`.
- Data reception:
  - To receive data over TCP, a `ServerSocket` is created and bound to a port number.
  - When a connection request is received, a new `Socket` is created for communication.
  - Data is received from the socket's input stream using methods like `InputStream.read(byte[])`.

2. UdP (User Datagram Protocol)

- UDP provides a connectionless, unreliable communication between two devices.
- In Java, UDP communication is typically implemented using the `DatagramSocket` class.
- Data transmission:

    - To send data over UDP, a `DatagramPacket` is created with the destination IP address and port number.

    - The data is then sent using the `DatagramSocket.send(DatagramPacket)` method.

  - Data reception:

    - To receive data over UDP, a `DatagramSocket` is created and bound to a port number.

    - Data is received into a `DatagramPacket` using the `DatagramSocket.receive(DatagramPacket)` method.

Here's a simple example illustrating UDP communication in Java:

```java
import java.net.*;

public class UDPServer {
    public static void main(String[] args) {
        try {
            DatagramSocket socket = new DatagramSocket(9876);
            byte[] buf = new byte[256];
            DatagramPacket packet = new DatagramPacket(buf, buf.length);

            socket.receive(packet);

            String received = new String(packet.getData(), 0, packet.getLength());
            System.out.println("Received: " + received);

            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

And the corresponding UDP client:

```java
import java.net.*;

public class UDPClient {
    public static void main(String[] args) {
        try {
            DatagramSocket socket = new DatagramSocket();
            InetAddress address = InetAddress.getByName("localhost");
            byte[] buf = "Hello, UDP!".getBytes();

            DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 9876);
            socket.send(packet);

            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

These examples demonstrate basic data transmission and reception using UDP in Java. Similar principles apply to TCP, but with the use of `Socket` and `ServerSocket` classes for communication.

10.Evaluate the advantages and disadvantages of using TCP and UDP protocols in Java networking for different types of applications

Advantages of TCP

*It is reliable for maintaining a connection between Sender and Receiver.

*It is responsible for sending data in a particular sequence.

*Its operations are not dependent on OS.

*It allows and supports many routing protocols.

*It can reduce the speed of data based on the speed of the receiver.

Disadvantages of TCP

*It is slower than UDP and it takes more bandwidth.

*Slower upon starting of transfer of a file.

*Not suitable for LAN and PAN Networks.

*It does not have a multicast or broadcast category.

*It does not load the whole page if a single data of the page is missing

Advantages of UDP

*It does not require any connection for sending or receiving data.

*Broadcast and Multicast are available in UDP.

*UDP can operate on a large range of networks.

*UDP has live and real-time data.

*UDP can deliver data if all the components of the data are not complete.

Disadvantages of UDP

*We can not have any way to acknowledge the successful transfer of data.

*UDP cannot have the mechanism to track the sequence of data.

*UDP is connectionless, and due to this, it is unreliable to transfer data.

*In case of a Collision, UDP packets are dropped by Routers in comparison to TCP.

*UDP can drop packets in case of detection of errors.

11.Evaluate the advantages and disadvantages of using thread priorities and state transitions in Java for performance tuning and resource allocation.

Advantages:

Performance Tuning: Thread priorities allow you to influence the order in which threads are scheduled by the JVM. This can be useful for giving higher priority to critical tasks, potentially improving overall system responsiveness.

Resource Allocation: By using thread priorities, you can allocate system resources more effectively. For example, you can allocate more CPU time to threads with higher priorities, ensuring that critical tasks are completed in a timely manner.

Responsiveness: Thread priorities can help improve the responsiveness of an application by allowing critical tasks to be processed quickly, even in the presence of other non-critical tasks.

Disadvantages:

Platform Dependency: The behavior of thread priorities and state transitions can vary between different Java Virtual Machine (JVM) implementations and operating systems. This can lead to non-deterministic behavior, making it difficult to predict the performance of an application.

Potential for Starvation: Giving high priorities to some threads can potentially starve lower-priority threads, leading to fairness issues in resource allocation.

Complexity: Managing thread priorities and state transitions can add complexity to your code, making it harder to understand and maintain.

12. Assess the significance of understanding and managing thread states and life cycles for efficient and responsive multithreaded Java applications?

Understanding and managing thread states and life cycles is crucial for developing efficient and responsive multithreaded Java applications. Here's why:

1. Efficiency:By understanding thread states and life cycles, developers can ensure that threads are used efficiently. This includes creating threads only when necessary, avoiding unnecessary context switches, and managing thread priorities appropriately.

2. Responsiveness:Managing thread states and life cycles effectively can help improve the responsiveness of an application. For example, by using techniques such as thread pooling, developers can ensure that threads are available to handle user requests promptly.

3. Resource Management: Thread states and life cycles are closely related to resource management. By managing thread states effectively, developers can avoid issues such as resource leaks and contention, which can degrade performance.

4. Debugging and Troubleshooting: Understanding thread states and life cycles is essential for debugging and troubleshooting multithreaded applications. It allows developers to identify issues such as deadlocks, livelocks, and thread starvation, and to implement solutions to address these problems.

5. Scalability: Proper management of thread states and life cycles is crucial for ensuring that an application can scale to handle increasing loads. By understanding the behavior of threads under different conditions, developers can design applications that can effectively utilize available resources.

13.Create a Java class that encapsulates complex data handling operations using different generic collections in Java?

Here's an example Java class that encapsulates complex data handling operations using different generic collections:

```java
import java.util.*;

public class DataHandler<T> {
    private List<T> list;
    private Set<T> set;
    private Map<Integer, T> map;

    public DataHandler() {
      list = new ArrayList<>();
      set = new HashSet<>();
      map = new HashMap<>();
    }

    public void addToList(T item) {
      list.add(item);
    }

    public void addToSet(T item) {
      set.add(item);
    }

    public void addToMap(Integer key, T value) {
      map.put(key, value);
    }

    public List<T> getList() {
      return list;
    }

    public Set<T> getSet() {
      return set;
    }

    public Map<Integer, T> getMap() {
      return map;
```

```
    }

    public void processData() {
        // Perform complex data handling operations here
        System.out.println("Processing data...");
    }

    public static void main(String[] args) {
        DataHandler<String> handler = new DataHandler<>();
        handler.addToList("One");
        handler.addToSet("Two");
        handler.addToMap(1, "Three");

        handler.processData();

        System.out.println("List: " + handler.getList());
        System.out.println("Set: " + handler.getSet());
        System.out.println("Map: " + handler.getMap());
    }
}
```

In this example, the `DataHandler` class encapsulates operations to add elements to a list, set, and map, as well as a method to process the data. The class uses generic collections to handle different types of data, making it flexible and reusable.

14.Create a Java class that encapsulates complex file I/O operations using both byte-oriented and character-oriented streams?

Here's an example Java class that encapsulates complex file I/O operations using both byte-oriented and character-oriented streams:

```java
import java.io.*;

public class FileIOHandler {
    private File file;

    public FileIOHandler(String filePath) {
        file = new File(filePath);
    }

    public void writeBytes(byte[] data) throws IOException {
        try (FileOutputStream fos = new FileOutputStream(file)) {
            fos.write(data);
        }
    }

    public byte[] readBytes() throws IOException {
        byte[] buffer = new byte[(int) file.length()];
        try (FileInputStream fis = new FileInputStream(file)) {
            fis.read(buffer);
        }
        return buffer;
    }

    public void writeChars(String data) throws IOException {
        try (FileWriter fw = new FileWriter(file)) {
            fw.write(data);
        }
    }
```

```java
public String readChars() throws IOException {
    StringBuilder sb = new StringBuilder();
    try (FileReader fr = new FileReader(file)) {
        int c;
        while ((c = fr.read()) != -1) {
            sb.append((char) c);
        }
    }
    return sb.toString();
}

public static void main(String[] args) {
    try {
        FileIOHandler handler = new FileIOHandler("test.txt");

        // Writing bytes to file
        byte[] bytes = {65, 66, 67, 68, 69};
        handler.writeBytes(bytes);

        // Reading bytes from file
        byte[] readBytes = handler.readBytes();
        System.out.println("Bytes read: " + new String(readBytes));

        // Writing characters to file
        handler.writeChars("Hello, world!");

        // Reading characters from file
        String readChars = handler.readChars();
        System.out.println("Characters read: " + readChars);
    } catch (IOException e) {
        e.printStackTrace();
        }
    }
}
```

In this example, the `FileIOHandler` class encapsulates methods for writing and reading both bytes and characters to/from a file. It uses `FileOutputStream`, `FileInputStream`, `FileWriter`, and `FileReader` to handle byte-oriented and character-oriented streams, respectively.

6 MARKS

1. Explain about Thread States and Life Cycles

In the context of multithreading, thread states and life cycles refer to the various stages a thread goes through from its creation to its termination. Understanding these states is crucial for managing threads effectively in a concurrent environment.

Thread States:

1. New:

- When a thread is created but has not yet started.
- At this stage, the thread has been instantiated but hasn't begun execution.

2. Runnable/Ready:

- The thread is ready to run and waiting for its turn to be picked by the scheduler.

- This state includes threads waiting for CPU time and those that are currently executing.

3. Blocked/Waiting:
   - The thread is waiting for a particular event or condition to occur before it can proceed.
   - This could be waiting for I/O operations, synchronization locks, or other signals.

4. Timed Waiting:
   - Similar to Blocked state but with a time limit.
   - Threads in this state are waiting for a specific amount of time before they can proceed.

5. Terminated
   - The thread has finished executing or has been explicitly terminated.
   - Once in this state, a thread cannot transition to any other state.

Thread Life Cycle:

1. Creation:
   - In this stage, the thread is instantiated by the program.
   - Resources such as memory space and a unique thread identifier are allocated.

2. Ready:
   - After creation, the thread enters the ready state, indicating it's prepared to run.
   - It's waiting for the scheduler to allocate CPU time.

3. Running:
   - The scheduler selects a thread from the pool of ready threads and executes it.
   - During execution, the thread moves between running, blocked, and timed waiting states depending on its behavior.

4. Waiting/Blocked:
   - When a thread needs to wait for an event (e.g., I/O operation, lock acquisition), it enters this state.

- It remains here until the event occurs.

5. Timed Waiting:

   - Similar to waiting, but with a time constraint.

   - The thread waits for a specific duration before it resumes execution.

6. Termination:

   - Once a thread finishes executing its task or is explicitly terminated by the program, it enters the termination state.

   - Resources associated with the thread are released, and it can no longer be scheduled for execution.

Transitions between States:

- Threads transition between states based on various factors such as resource availability, external events, and synchronization mechanisms.

- For example, a thread may move from the running state to the blocked state when it requests access to a shared resource held by another thread.

- Thread state transitions are managed by the operating system's scheduler and are influenced by thread priorities, synchronization primitives, and other scheduling policies.

Understanding thread states and life cycles is essential for developing robust concurrent programs and effectively managing resources in multithreaded environments.

2.Describe the role of keywords try, throw and catch in exception handling.

In most programming languages, including Java and C++, exception handling is a critical feature that allows developers to gracefully manage errors and unexpected situations that may occur during program execution. The `try`, `throw`, and `catch` keywords are fundamental components of exception handling mechanisms. Here's an explanation of each:

1. try:

   - The `try` block is used to enclose a section of code where exceptions might occur.

   - It essentially defines the scope within which exceptions will be monitored.

- When an exception is thrown within the `try` block, the control flow is transferred to the corresponding `catch` block.

- Multiple `catch` blocks can follow a single `try` block to handle different types of exceptions.

2. throw:

- The `throw` keyword is used to explicitly throw an exception within the program.

- It typically appears within conditional statements or other control flow structures where an error condition is detected.

- When `throw` is executed, it generates an exception object and passes control to the nearest enclosing `try` block to handle the exception.

3. catch:

- The `catch` block follows a `try` block and is used to handle exceptions thrown within that `try` block.

- Each `catch` block specifies the type of exception it can handle.

- When an exception occurs in the `try` block, the runtime system looks for the appropriate `catch` block that matches the type of the thrown exception.

- If a matching `catch` block is found, the code within that block is executed to handle the exception. If not, the exception propagates up the call stack to higher-level `try` blocks or terminates the program if not caught at all.

Example (in Java):

```java
try {
   // Code that might throw an exception
   int result = divide(10, 0); // This could throw an ArithmeticException
   System.out.println("Result: " + result); // This line will not execute if an exception occurs
} catch (ArithmeticException e) {
   // Handling specific exception type
   System.err.println("Arithmetic Exception: " + e.getMessage());
} catch (Exception e) {
   // Handling other exceptions
```

```
    System.err.println("Exception occurred: " + e.getMessage());
}

// Method that throws an exception
public static int divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw new ArithmeticException("Denominator cannot be zero");
    }
    return numerator / denominator;
}
```

In this example, the `try` block contains code that divides two numbers, which may result in an `ArithmeticException` if the denominator is zero. If such an exception occurs, control is transferred to the corresponding `catch` block, where the exception is handled appropriately. If any other exception occurs that is not specifically caught by the first `catch` block, it will be caught by the more general `catch (Exception e)` block.

3.Describe on Thread Synchronization

Thread synchronization is a fundamental concept in concurrent programming, especially in environments where multiple threads access shared resources simultaneously. Without proper synchronization mechanisms, concurrent access to shared data can lead to race conditions, data corruption, and other unpredictable behaviors. Thread synchronization ensures that threads coordinate their activities to avoid such issues. Here's a discussion on thread synchronization:

1. Shared Resources:

- Shared resources refer to data structures, variables, or objects that are accessible by multiple threads.

- Accessing shared resources concurrently without synchronization can lead to inconsistencies and errors in the program.

2. Race Conditions:

- Race conditions occur when the outcome of a program depends on the timing or interleaving of multiple threads' execution.

- They can lead to unpredictable behavior and incorrect results due to non-deterministic execution order.

3. Critical Sections:
   - Critical sections are parts of the code where shared resources are accessed and modified.
   - It's crucial to ensure that only one thread can execute a critical section at a time to maintain data integrity.

4. Synchronization Mechanisms:
   - Locks/Mutexes:
     - Locks (mutexes) are synchronization primitives that allow threads to acquire exclusive access to a shared resource.
     - Threads must acquire the lock before entering a critical section and release it when they're done.
   - Semaphores:
     - Semaphores are counters used to control access to a resource.
     - They can be used to limit the number of threads accessing a resource simultaneously.
   - Monitors:
     - Monitors are higher-level synchronization constructs that encapsulate shared data and methods to manipulate that data.
     - They provide mutual exclusion and condition synchronization within a single construct.
   - Atomic Operations:
     - Atomic operations are operations that are executed as a single, indivisible unit.
     - They ensure that certain operations on shared data are performed atomically, without interruption.

5. Thread Safety:
   - Ensuring thread safety involves designing classes and methods in a way that allows them to be used safely in a multithreaded environment.
   - Thread-safe data structures and algorithms are designed to handle concurrent access gracefully, typically by employing synchronization mechanisms internally.

6. Deadlocks and Starvation:

- Inefficient use of synchronization mechanisms can lead to deadlocks, where threads are unable to proceed because they're waiting for resources held by other threads.

- Starvation occurs when a thread is unable to gain access to a shared resource indefinitely due to unfair scheduling.

7. Performance Considerations:

- Synchronization mechanisms add overhead to the program execution.

- Careful consideration must be given to the choice of synchronization technique to minimize overhead while ensuring correctness.

8. Higher-Level Concurrency Constructs:

- Many programming languages and libraries provide higher-level concurrency constructs and abstractions that encapsulate synchronization details.

- Examples include concurrent data structures, parallel processing frameworks, and actor-based models.

In summary, thread synchronization is essential for ensuring the correctness and reliability of concurrent programs by coordinating access to shared resources and preventing race conditions and other concurrency issues. Proper understanding and implementation of synchronization mechanisms are crucial for developing robust multithreaded applications.

4. Show the use of Java Networking.

Networking supplements a lot of power to simple programs. With networks, a single program can regain information stored in millions of computers positioned anywhere in the world. Java is the leading programming language composed from scratch with networking in mind. Java Networking is a notion of combining two or more computing devices together to share resources.

All the Java program communications over the network are done at the application layer. The java.net package of the J2SE APIs comprises various classes and interfaces that execute the low-level communication features, enabling the user to formulate programs that focus on resolving the problem.

The java.net package of the Java programming language includes various classes and interfaces that provide an easy-to-use means to access network resources. Other than classes and interfaces, the java.net package also provides support for the two well-known network protocols.

These are:

Transmission Control Protocol (TCP)

TCP or Transmission Control Protocol allows secure communication between different applications. TCP is a connection-oriented protocol which means that once a connection is established, data can be transmitted in two directions. This protocol is typically used over the Internet Protocol. Therefore, TCP is also referred to as TCP/IP. TCP has built-in methods to examine for errors and ensure the delivery of data in the order it was sent, making it a complete protocol for transporting information like still images, data files, and web pages.

User Datagram Protocol (UDP)

UDP or User Datagram Protocol is a connection-less protocol that allows data packets to be transmitted between different applications. UDP is a simpler Internet protocol in which error-checking and recovery services are not required. In UDP, there is no overhead for opening a connection, maintaining a connection, or terminating a connection. In UDP, the data is continuously sent to the recipient, whether they receive it or not.

5. With the help of an example, Explain multithreading by extending Thread class ?

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

Threads can be created by using two mechanisms :

Extending the Thread class

Implementing the Runnable Interface

Thread creation by extending the Thread class

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

```
// Java code for thread creation by extending

// the Thread class

class MultithreadingDemo extends Thread {
        public void run()
        {
                try {
                        // Displaying the thread that is running
                        System.out.println(
                                "Thread " + Thread.currentThread().getId()
                                + " is running");
                }
                catch (Exception e) {
                        // Throwing an exception
                        System.out.println("Exception is caught");
                }
        }
}

// Main Class
public class Multithread {
        public static void main(String[] args)
        {
                int n = 8; // Number of threads
                for (int i = 0; i < n; i++) {
                        MultithreadingDemo object
                                = new MultithreadingDemo();
                        object.start();
                }
        }
}
```

6.Show how to make GUI Development using SWING

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

public class SwingExample {
  public static void main(String[] args) {
    // Create a frame (window)
    JFrame frame = new JFrame("Swing Example");

    // Set frame size
    frame.setSize(300, 200);

    // Set frame layout
    frame.setLayout(new FlowLayout());

    // Create a label
    JLabel label = new JLabel("Hello, Swing!");

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

    // Add action listener to the button
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // Change the text of the label when button is clicked
        label.setText("Button Clicked!");
      }
    });

    // Add label and button to the frame
    frame.add(label);
    frame.add(button);
```

```
        // Set frame visibility
        frame.setVisible(true);

        // Set default close operation
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

This code creates a simple Swing GUI application with a frame containing a label and a button. When the button is clicked, it changes the text of the label. Here's a breakdown of the key components:

JFrame: Represents the main window or frame of the application.

JLabel: Displays text or an image. In this example, it displays "Hello, Swing!" initially.

JButton: Represents a button component. It has an action listener attached to it to handle button clicks.

ActionListener: Interface for handling action events, such as button clicks. In this example, we use an anonymous inner class to implement it.

FlowLayout: Layout manager that arranges components in a left-to-right flow.

To run this code, simply compile it and execute the SwingExample class. You should see a window with a label and a button. When you click the button, the text of the label changes.

7.In Java, is exception handling implicit or explicit or both? Explain with the help of example Java programs .

In Java, exception handling can be both implicit and explicit, depending on the context and the type of exceptions being handled.

Implicit Exception Handling:

Implicit exception handling occurs when the programmer relies on the default exception handling mechanism provided by the Java runtime environment. This typically involves propagating exceptions up the call stack until they are caught by an appropriate exception handler, either within the application or at the top-level.

Here's an example demonstrating implicit exception handling:

```java
public class ImplicitExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            // Code that may throw an exception
            int result = 10 / 0; // This will throw an ArithmeticException
        } catch (ArithmeticException e) {
            // Exception caught implicitly by the default exception handling mechanism
            System.out.println("An arithmetic exception occurred: " + e.getMessage());
        }
    }
}
```

In this example, the division by zero operation will throw an `ArithmeticException`. This exception is implicitly caught by the default exception handling mechanism provided by Java. The program will print the error message and stack trace to the console.

Explicit Exception Handling:

Explicit exception handling occurs when the programmer explicitly specifies how to handle exceptions using `try`, `catch`, and `finally` blocks.

Here's an example demonstrating explicit exception handling:

```java
public class ExplicitExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            // Code that may throw an exception
            int result = divide(10, 0); // This will throw an ArithmeticException
            System.out.println("Result: " + result); // This line will not execute if an exception occurs
        } catch (ArithmeticException e) {
            // Explicitly catching the ArithmeticException
            System.out.println("An arithmetic exception occurred: " + e.getMessage());
        }
    }

    // Method that throws an exception
    public static int divide(int numerator, int denominator) {
        if (denominator == 0) {
            throw new ArithmeticException("Denominator cannot be zero");
        }
        return numerator / denominator;
    }
}
```

In this example, we explicitly use a `try-catch` block to handle the `ArithmeticException` thrown by the `divide` method. We catch the exception and print a custom error message. Without this explicit handling, the exception would propagate up the call stack until it's caught or until it reaches the default exception handler.

Conclusion:

Java supports both implicit and explicit exception handling mechanisms. While implicit handling relies on the default exception handling provided by the Java runtime environment, explicit handling gives programmers more control over how exceptions are handled within their code. Both approaches are valid and have their use cases depending on the requirements of the application.

8.Implementing Run able interface and extending Thread, which method you prefer for multithreading and why?

Both implementing the `Runnable` interface and extending the `Thread` class are common ways to achieve multithreading in Java. Each approach has its advantages and use cases. Let's discuss both:

Implementing Runnable Interface:

```java
public class MyRunnable implements Runnable {
    public void run() {
        // Code to be executed by the thread
        System.out.println("Thread executing using Runnable interface");
    }

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start();
    }
}
```

Advantages:

1. **Flexibility:** Implementing `Runnable` allows you to extend another class if needed, as Java does not support multiple inheritance.

2. **Better Object-Oriented Design:** Promotes better object-oriented design principles, as it separates the task from the thread.

3. **Reusability:** You can reuse the same `Runnable` instance with multiple threads.

Extending Thread Class:

```java
public class MyThread extends Thread {
    public void run() {
        // Code to be executed by the thread
        System.out.println("Thread executing by extending Thread class");
    }

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start();
    }
}
```

Advantages:

1. **Simplicity:** Extending `Thread` is simpler and requires less boilerplate code, as you directly override the `run` method.

2. **Convenience:** It may be more convenient for simple cases where you only need to define the thread's behavior.

Preference and Recommendations:

- **Prefer Implementing Runnable:**
  - Promotes better design practices such as separation of concerns and reuse.
  - Avoids limitations imposed by single inheritance in Java.
  - Allows for better encapsulation and flexibility.
- **Extending Thread for Simplicity:**
  - If you have a simple use case and don't need the added flexibility provided by implementing `Runnable`.
  - For quick prototyping or when you don't need to extend other classes.

In conclusion, while both approaches have their merits, implementing the `Runnable` interface is generally preferred due to its better support for object-oriented design principles and flexibility. However, for simpler cases or quick prototyping, extending the `Thread` class may be more convenient.

9.Implementing Run able interface and extending Thread, which method you prefer for multithreading and why?

When it comes to choosing between implementing the `Runnable` interface and extending the `Thread` class for multithreading in Java, the preferred method is implementing the `Runnable` interface. Here's why:

Implementing Runnable Interface:

1. **Better Object-Oriented Design:**

   - Implementing `Runnable` promotes better object-oriented design principles by separating the task (the `Runnable` object) from the thread (the `Thread` object).

   - This separation of concerns makes your code more modular, easier to understand, and maintainable.

2. **Flexibility:**

   - Java does not support multiple inheritance, so implementing `Runnable` allows you to extend another class if needed, whereas extending `Thread` consumes your one chance to extend a class.

   - You can pass the same `Runnable` instance to multiple threads, promoting reusability.

3. **Encapsulation:**

   - By implementing `Runnable`, you encapsulate the task that needs to be executed in its own class, making it easier to manage and test.

4. **Promotes Composition over Inheritance:**

   - It follows the principle of composition over inheritance, which is generally considered a better design practice in object-oriented programming.

Extending Thread Class:

1. **Simplicity:**

   - Extending `Thread` may be simpler for very simple cases where you only need to define the behavior of the thread.

   - It requires less boilerplate code since you directly override the `run` method of the `Thread` class.

2. **Convenience:**

   - It might be more convenient for quick prototyping or when you have a very simple use case.

Conclusion:

While extending the `Thread` class might seem simpler initially, implementing the `Runnable` interface is generally preferred for its better support of object-oriented design principles, flexibility, and encapsulation. It promotes modular and maintainable code, allows for better composition, and facilitates reuse of the task logic. Therefore, unless you have a very compelling reason to extend `Thread`, such as simplicity for a quick prototype, implementing `Runnable` is the recommended approach for multithreading in Java.

10.Analyze the steps for JAVA GUI

Creating a graphical user interface (GUI) in Java typically involves several steps. Here's an overview of the steps involved in building a basic GUI using Java:

1. Import Necessary Packages:

Import the necessary packages from the `javax.swing` and `java.awt` libraries. These packages contain classes and interfaces for creating GUI components and managing the GUI layout.

```java
import javax.swing.*;

import java.awt.*;
```

2. Create a JFrame:

Create a `JFrame` object, which represents the main window or frame of the GUI application.

```java
JFrame frame = new JFrame("My GUI Application");
```

3. Set Frame Properties:

Set properties of the frame, such as size, default close operation, and layout manager.

```java
```

frame.setSize(400, 300);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setLayout(new FlowLayout()); // or other layout manager

4. Create GUI Components:

Create various GUI components such as buttons, labels, text fields, etc.

```java

JButton button = new JButton("Click Me");

JLabel label = new JLabel("Hello, World!");

JTextField textField = new JTextField(20);

// Add more components as needed

```

5. Add Components to Frame:

Add the created components to the frame using the `add()` method.

```java

frame.add(button);

frame.add(label);

frame.add(textField);

// Add more components as needed

```

6. Set Event Listeners:

Set event listeners (e.g., ActionListener, MouseListener) on interactive components to handle user input and interactions.

```java

button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Code to be executed when the button is clicked
        label.setText("Button Clicked!");
    }
});
// Add more event listeners as needed
```

7. Display Frame:
Set the frame's visibility to `true` to display the GUI on the screen.

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

Full Example:
Here's a complete example combining all the steps:

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

public class MyGUI {
    public static void main(String[] args) {
        JFrame frame = new JFrame("My GUI Application");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());

        JButton button = new JButton("Click Me");
        JLabel label = new JLabel("Hello, World!");
        JTextField textField = new JTextField(20);

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                label.setText("Button Clicked!");
            }
        });

        frame.add(button);
        frame.add(label);
        frame.add(textField);

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

Conclusion:

These steps provide a basic framework for creating a GUI in Java using Swing. You can customize and extend this framework to build more complex and interactive GUI applications based on your requirements. Additionally, you can explore other layout managers, event listeners, and GUI components provided by Swing to create rich and responsive user interfaces.

11.Evaluate the performance of Thread Synchronization

Evaluating the performance of thread synchronization involves measuring the overhead introduced by synchronization mechanisms in a multithreaded environment. Here are some factors to consider when evaluating the performance of thread synchronization:

1. Overhead:

Thread synchronization introduces overhead in terms of additional processing time and memory consumption. It's essential to measure this overhead to understand its impact on overall performance.

2. Scalability:

Evaluate how well the synchronization mechanism scales as the number of threads increases. Some synchronization techniques may become less efficient or even introduce contention as the number of threads grows.

3. Contentions:

Contention occurs when multiple threads contend for access to a shared resource simultaneously. Measure the frequency and impact of contentions on overall performance. High contention can lead to decreased throughput and increased latency.

4. Locking Granularity:

The granularity of locking affects performance. Fine-grained locking may reduce contention but can introduce overhead due to frequent lock acquisitions and releases. Coarse-grained locking may reduce overhead but increase contention.

5. Locking Strategies:

Different locking strategies, such as intrinsic locks (synchronized blocks/methods) and explicit locks (java.util.concurrent locks), have varying performance characteristics. Evaluate the performance of different locking strategies to choose the most suitable one for your application.

6. Locking Duration:

Measure the duration for which threads hold locks. Long-held locks can lead to increased contention and reduced concurrency, impacting performance.

7. Lock-Free Techniques:

Consider using lock-free or non-blocking synchronization techniques, such as atomic operations, compare-and-swap (CAS), and concurrent data structures, to minimize contention and improve performance in highly concurrent scenarios.

8. Profiling Tools:

Use profiling tools and performance monitoring utilities to identify bottlenecks and hotspots in your multithreaded application. These tools can help pinpoint areas where synchronization overhead is significant and optimize performance accordingly.

9. Benchmarking:

Perform benchmarking tests under various workloads and concurrency levels to evaluate the performance of synchronization mechanisms comprehensively. Compare the performance of synchronized and unsynchronized versions of your code to quantify the impact of synchronization overhead.

10. Platform and Environment:

Consider the platform and environment in which your application runs. Performance characteristics may vary depending on factors such as the operating system, hardware architecture, JVM implementation, and runtime parameters.

Conclusion:

Evaluating the performance of thread synchronization requires careful measurement and analysis of various factors, including overhead, scalability, contention, locking granularity, locking strategies, locking duration, profiling data, benchmarking results, and the platform/environment. By understanding these factors and optimizing synchronization mechanisms accordingly, you can achieve better performance and scalability in your multithreaded applications.

12.Compare I/O Streams and Object Serialization.

I/O Streams:	**Object Serialization:**
I/O streams handle raw data in bytes or characters, providing low-level access to input/output sources.	**Object serialization deals specifically with Java objects, providing high-level support for object persistence and communication.**
I/O streams offer more flexibility in terms	**Object serialization is tailored specifically**

of data manipulation and processing, supporting a wide range of input/output sources and formats.	**for serializing Java objects, providing a convenient way to handle object serialization/deserialization tasks.**
I/O streams offer better performance for simple data transfer operations, especially for large volumes of data	**Object serialization provides convenience and ease of use for handling object serialization tasks, but may incur performance overhead, particularly for complex object graphs.**
● I/O streams provide a way to transfer data between a Java program and an external source, such as a file, network connection, or another program. ● They facilitate reading from and writing to different types of data sources, such as bytes, characters, or objects	● Object serialization is a mechanism in Java that allows objects to be converted into a stream of bytes, which can then be written to a file, transmitted over a network, or stored in a database. ● It provides a way to persist object state, transfer objects between different Java applications, and implement distributed computing.
● Java provides two main types of I/O streams: byte streams and character streams. ● Byte streams (`InputStream` and `OutputStream`) are used for reading and writing binary data, such as files, sockets, and byte arrays. ● Character streams (`Reader` and `Writer`) are used for reading and writing textual data, automatically handling character encoding and decoding.	● Object serialization is used when you need to save the state of an object to disk (serialization) or transmit an object across a network (marshalling) or between different Java Virtual Machines (JVMs) in a distributed system. ● It's commonly used in scenarios such as saving and loading application state, caching objects, implementing remote method invocation (RMI), and messaging systems.
I/O streams are commonly used for tasks such as reading/writing files, communicating over network sockets, parsing input data, and interacting with other processes.	● Object serialization can incur performance overhead due to the process of converting objects into byte streams and vice versa. ● Serialization/deserialization process may involve reflection, class

	metadata, and additional processing, which can impact performance, especially for large or complex object graphs.

10 MARKS

1.Construct the GUI Development using SWING?

To create a simple GUI in Java, you can use the JFrame class, imported with import javax.swing.JFrame;. Once imported, you can create a new customized instance by setting the properties, like the size, .setSize([dimensions]);, and visibility with, .setVisible([true or false]);. This class provides the basic functionality to create a window on your screen.

Here’s a basic example:

```
import javax.swing.JFrame;

public class Main {
   public static void main(String[] args) {
      JFrame frame = new JFrame();
      frame.setSize(400, 400);
      frame.setVisible(true);
   }
}

// Output:
// A simple window of size 400x400 is displayed on the screen.
```

In this example, we import the JFrame class from the javax.swing package. We then create a new JFrame object, set its size to 400×400 pixels, and make it visible. This results in a simple window being displayed on the screen.

This is just a basic way to use Java Swing to create a GUI, but there's much more to learn about creating and manipulating GUIs in Java Swing. Continue reading for more detailed information and advanced usage scenarios.

Basic Use of Java Swing

Java Swing is a powerful library for creating GUIs in Java. For beginners, understanding the basics is the first step to mastering its use. Let's start by creating a simple GUI using Swing.

Step 1: Importing Swing Libraries

The first step is to import the necessary Swing libraries into your Java code. For a basic GUI, we need the JFrame class, which is part of the javax.swing package.

```
import javax.swing.JFrame;
```

Java

Step 2: Creating a New JFrame

Next, we create a new JFrame object. This represents a window in our GUI.

```
JFrame frame = new JFrame();
```

Java

Step 3: Setting the Frame Size

We then set the size of the frame using the setSize method. This method takes two parameters: the width and the height of the window in pixels.

```
frame.setSize(600, 600);
```

Java

Step 4: Making the Frame Visible

Finally, we make the frame visible using the setVisible method. This method takes a boolean value. When set to true, the frame is displayed.

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

Java

Putting it all together, we have:

```
import javax.swing.JFrame;

public class Main {
  public static void main(String[] args) {
    JFrame frame = new JFrame();
    frame.setSize(600, 600);
    frame.setVisible(true);
  }
}

// Output:
// A simple window of size 600x600 is displayed on the screen.
```

Java

This is a simple example of using Java Swing to create a GUI. However, a real-world application will have more complex requirements. In the next section, we'll look at some more advanced uses of Swing.

2.Apply the concept of Thread States and Life Cycles?

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1.New State

2.Runnable State

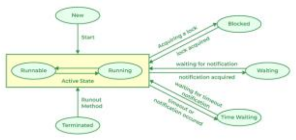
3.Blocked State

4.Waiting State

5.Timed Waiting State

6.Terminated State

The diagram shown below represents various states of a thread at any instant in time.



Life Cycle of a Thread

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

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

2.Runnable State: A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.

*A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

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

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

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

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

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

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

3.Explain the methods of Java Networking?

- Networking supplements a lot of power to simple programs. With networks, a single program can regain information stored in millions of computers positioned anywhere in the world. Java is the leading programming language composed from scratch with networking in mind. Java Networking is a notion of combining two or more computing devices together to share resources.

- All the Java program communications over the network are done at the application layer. The java.net package of the J2SE APIs comprises various classes and interfaces that execute the low-level communication features, enabling the user to formulate programs that focus on resolving the problem.

- Common Network Protocols

As stated earlier, the java.net package of the Java programming language includes various classes and interfaces that provide an easy-to-use means to access network resources. Other than classes and interfaces, the java.net package also provides support for the two well-known network protocols. These are:

1.Transmission Control Protocol (TCP) – TCP or Transmission Control Protocol allows secure communication between different applications. TCP is a connection-oriented protocol which means that once a connection is established, data can be transmitted in two directions. This protocol is typically used over the Internet Protocol. Therefore, TCP is also referred to as TCP/IP. TCP has built-in methods to examine for errors and ensure the delivery of data in the order it was sent, making it a complete protocol for transporting information like still images, data files, and web pages.

2.User Datagram Protocol (UDP) – UDP or User Datagram Protocol is a connection-less protocol that allows data packets to be transmitted between different applications. UDP is a simpler Internet protocol in which error-checking and recovery services are not required. In UDP, there is no overhead for opening a connection, maintaining a connection, or terminating a connection. In UDP, the data is continuously sent to the recipient, whether they receive it or not.

- Java Networking Terminology

In Java Networking, many terminologies are used frequently. These widely used Java Networking Terminologies are given as follows:

- IP Address – An IP address is a unique address that distinguishes a device on the internet or a local network. IP stands for "Internet Protocol." It comprises a set of rules governing the format of data sent via the internet or local network. IP Address is referred to as a logical address that can be modified. It is composed of octets. The range of each octet varies from 0 to 255.

Range of the IP Address – 0.0.0.0 to 255.255.255.255

For Example – 192.168.0.1

- Port Number – A port number is a method to recognize a particular process connecting internet or other network information when it reaches a server. The port number is used to identify different applications uniquely. The port number behaves as a communication endpoint among applications. The port number is correlated with the IP address for transmission and communication among two applications. There are 65,535 port numbers, but not all are used every day.

- Protocol – A network protocol is an organized set of commands that define how data is transmitted between different devices in the same network. Network protocols are the reason through which a user can easily communicate with people all over the world and thus play a critical role in modern digital communications. For Example – TCP, FTP, POP, etc.

- MAC Address – MAC address stands for Media Access Control address. It is a bizarre identifier that is allocated to a NIC (Network Interface Controller/ Card). It contains a 48 bit or 64-bit address, which is combined with the network adapter. MAC address can be in hexadecimal composition. In simple words, a MAC address is a unique number that is used to track a device in a network.

- Socket – A socket is one endpoint of a two-way communication connection between the two applications running on the network. The socket mechanism presents a method of inter-process communication (IPC) by setting named contact points between which the communication occurs. A socket is tied to a port number so that the TCP layer can recognize the application to which the data is intended to be sent.

- Connection-oriented and connection-less protocol – In a connection-oriented service, the user must establish a connection before starting the communication. When the connection is established, the user can send the message or the information, and after this, they can release the connection. However, In connectionless protocol, the data is transported in one route from source to destination without verifying that the destination is still there or not or if it is ready to receive the message. Authentication is not needed in the connectionless protocol.

Example of Connection-oriented Protocol – Transmission Control Protocol (TCP)

Example of Connectionless Protocol – User Datagram Protocol (UDP)

4.Explain in detail about I/O Streams in JAVA?

Java I/O Streams

In Java, streams are the sequence of data that are read from the source and written to the destination.

An input stream is used to read data from the source. And, an output stream is used to write data to the destination.

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

Run Code

For example, in our first Hello World example, we have used System.out to print a string. Here, the System.out is a type of output stream.

Types of Streams

Depending upon the data a stream holds, it can be classified into:

- Byte Stream
- Character Stream

Byte Stream:

Byte stream is used to read and write a single byte (8 bits) of data.

All byte stream classes are derived from base abstract classes called InputStream and OutputStream.

To learn more, visit

- Java InputStream Class
- Java OutputStream Class

Character Stream

Character stream is used to read and write a single character of data.

All the character stream classes are derived from base abstract classes Reader and Writer.

5.Give a note on GUI Development.

GUI, which stands for Graphical User Interface, is a user-friendly visual experience builder for Java applications. It comprises graphical units like buttons, labels, windows, etc. via which users can connect with an application. Swing and JavaFX are two commonly used applications to create GUIs in Java.

Elements of GUI:

A GUI comprises an array of user interface elements. All these elements are displayed when a user is interacting with an application and they are as follows:

1. Input commands such as buttons, check boxes, dropdown lists and text fields.

2. Informational components like banners, icons, labels or notification dialogs.

3. Navigational units like menus, sidebars and breadcrumbs.

GUI in JAVA: Swing and JavaFX

As mentioned above, to create a GUI in Java, Swing and JavaFX are the most commonly used applications. Swing was designed with a flexible architecture to make the elements customizable and easy to plug-and-play which is why it is the first choice for java developers while creating GUIs.

As far as JavaFX is concerned, it consists of a totally different set of graphic components along with new features and terminologies.

Creating a GUI

The process of creating a GUI in Swing starts with creating a class that represents the main GUI. An article of this class acts as a container which holds all the other components to be displayed.

In most of the projects, the main interface article is a frame, i.e., the JFrame class in javax.swing package. A frame is basically a window which is displayed whenever a user opens an application on his/her computer. It has a title bar and buttons such as minimize, maximize and close along with other features.

The JFrame class consists of simple constructors such as JFrame() and JFrame(String). The JFrame() leaves the frame's title bar empty, whereas the JFrame(String) places the title bar to a specified text.

Apart from the title, the size of the frame can also be customized. It can be established by incorporating the setSize(int, int) method by inserting the width and height desired for the frame. The size of a frame is always designated in pixels.

For example, calling setSize(550,350) would create a frame that would be 550 pixels wide and 350 pixels tall.

Usually, frames are invisible at the time of their creation. However, a user can make them visible by using the frame's setVisible(boolean) method by using the word 'true' as an argument.

The following are the steps to create GUI in Java

STEP 1: The following code is to be copied into an editor

```
 import javax.swing.*;

class gui{

public static void main(String args[]){

    JFrame jframe = new JFrame("GUI Screen");   //create JFrame object

    jframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    jframe.setSize(400,400);        //set size of GUI screen

    jframe.setVisible(true);

}
}
```

STEP 2: Save and compile the code as mentioned above and then run it.

STEP 3: Adding buttons to the above frame. To create a component in Java, the user is required to create an object of that component's class. We have already understood the container class JFrame.

One such component to implement is JButton. This class represents the clickable buttons. In any application or program, buttons trigger user actions. Literally, every action begins with a click; like to close an application, the user would click on the close button.

A swing can also be inserted, which can feature a text, a graphical icon or a combination of both. A user can use the following constructors:

· JButton(String): This button is labelled with a specified text.

· JButton(Icon): This button is labelled with a graphical icon.

· JButton(String,Icon): This button is labelled with a combination of text and icon.

The following code is to be copied into an editor:

```
import javax.swing.*;

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

        JFrame jframe = new JFrame("GUI Screen");   //create JFrame object

        jframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        jframe.setSize(400,400);            //set size of GUI screen

        JButton pressButton = new JButton("Press");  //create JButton object

        jframe.getContentPane().add(pressButton);

        jframe.setVisible(true);

    }

}
```

STEP 4: The above is to be executed. A big button will appear on the screen.

STEP 5: A user can add two buttons to the frame as well. Copy the code given below into an editor.

```
import javax.swing.*;


class gui{


    public static void main(String args[]){
        JFrame jframe = new JFrame("GUI Screen");

        jframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        jframe.setSize(400,400);

        JButton firstButton = new JButton("First Button");  //create firstButton object

        JButton secondButton = new JButton("Second Button");  //create secondButton object

        jframe.getContentPane().add(firstButton);

        jframe.getContentPane().add(secondButton);

        jframe.setVisible(true);

    }

}
```

STEP 6: Save, compile and run the above code.

STEP 7: Unpredicted output = ? It means that the buttons are getting overlapped.

STEP 8: A user can create chat frames as well.