JAVA UNIT 4

1. What is Multithreading? List its Advantages.

Multithreading is a programming concept in Java that allows multiple threads to run concurrently within a single process. Each thread represents an independent path of execution, but all threads share the same memory space. This helps a program perform multiple tasks at the same time, improving efficiency and performance

Advantages of Multithreading:

1. **Improved CPU Utilization:**
   Multithreading allows better use of CPU resources by executing multiple tasks simultaneously. Idle CPU time is reduced because threads can run in parallel on multiple cores.
2. **Enhanced Responsiveness:**
   Applications remain responsive even when performing background tasks. For example, a user interface can remain active while data is being processed in another thread.
3. **Efficient Resource Sharing:**
   Threads share the same memory space, which reduces memory usage. This makes inter-thread communication faster compared to separate processes.
4. **Faster Execution:**
   Tasks are divided into smaller units and executed concurrently. This reduces overall execution time for complex applications.
5. **Better Application Performance:**
   Multithreading is useful in server applications where multiple clients are handled simultaneously. It improves scalability and throughput.

2. Differentiate between Process and Thread.

A **process** is an independent program under execution, while a **thread** is a lightweight sub-unit of a process. Multiple threads can exist within a single process and share its resources

1. **Definition:** A process is a program in execution with its own memory space. A thread is the smallest unit of execution within a process.
2. **Memory Sharing:** Processes have separate memory spaces, so communication is costly. Threads share the same memory, making communication faster.
3. **Creation Cost:** Creating a process is expensive in terms of system resources. Creating a thread is faster and requires fewer resources.
4. **Communication:** Inter-process communication is complex and slower. Inter-thread communication is easier because threads share memory.
5. **Failure Impact:** If a process crashes, other processes are unaffected. If one thread fails, it can affect the entire process.
6. **Scheduling:** Processes are scheduled independently by the OS. Threads are scheduled within the process, making context switching faster.

3. Explain Thread Life Cycle with Diagram / Explain Thread States.

The **thread life cycle** describes the various states a thread passes through from creation to termination. Java threads move between these states based on method calls and execution flow

1. **New (Newborn) State:**
   A thread is in the new state when it is created using the Thread class. At this stage, the start() method has not been called.

2. **Runnable State:**
   After start() is called, the thread enters the runnable state. The thread is ready to run and waits for CPU allocation.

3. **Running State:**
   In this state, the thread gets CPU time and executes its task. It remains running until it finishes or is preempted.

4. **Blocked / Waiting State:**
   A thread enters this state when it waits for resources or sleeps. Methods like sleep(), wait(), or join() cause this state.

5. **Dead (Terminated) State:**
   A thread enters the dead state after completing execution. Once terminated, it cannot be restarted.

4. Thread Creation Using Thread Class and Runnable Interface. Compare Both.

Threads in Java can be created using **Thread class** or **Runnable interface**. Both methods define the run() method, which contains the thread logic

Using Thread Class:

1. A class extends the Thread class and overrides the run() method.
2. The thread starts execution when start() is called.
3. This method is simple but restricts multiple inheritance.

Example:

```
class MyThread extends Thread {

 public void run() {

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

 }

}
```

Using Runnable Interface:

1. A class implements Runnable and defines run().
2. The Runnable object is passed to a Thread object.
3. This supports multiple inheritance and better design.

Example:

```
class MyRun implements Runnable {
 public void run() {
  System.out.println("Thread running");
 } }
```

Comparison:

1. Thread class uses inheritance, Runnable uses interface.
2. Runnable is more flexible and preferred in real applications.

5. Explain Main Thread and Its Properties with Example.

The **main thread** is the first thread that starts when a Java program begins execution. It controls the execution of the program and can create child threads

1. **Automatic Creation:** The main thread is created automatically by the JVM. It starts execution from the main() method.
2. **Thread Control:** The main thread can be controlled like other threads using Thread class methods. It can change priority or be paused.
3. **Accessing Main Thread:** The currentThread() method is used to get a reference to the main thread. This allows us to inspect its properties.
4. **Properties:** The main thread has a default priority of 5. It belongs to the main thread group.
5. **Program Termination:** When the main thread ends, the program terminates if no other threads are running.

Example Program:

```
public class MainThreadDemo {
 public static void main(String[] args) {
  Thread t = Thread.currentThread();
  System.out.println(t.getName());
 } }
```

6. Explain Thread Synchronization and Ways to Implement It in Java. Write Programs.

Thread synchronization is a mechanism in Java that ensures only one thread accesses a shared resource at a time. It prevents data inconsistency and race conditions when multiple threads modify shared data simultaneously

Why Synchronization is Needed

1. **Prevents Race Condition:**
   Without synchronization, multiple threads may update shared variables incorrectly. This leads to unpredictable and wrong output.

2. **Ensures Data Consistency:**
   Synchronization makes sure data remains accurate when accessed by many threads. Only one thread executes the critical section at a time.

Ways to Implement Synchronization

1. **Synchronized Method:**
   The entire method is locked so only one thread can execute it at a time. Other threads must wait until the lock is released.

2. **Synchronized Block:**
   Only a specific block of code is synchronized. This improves performance by locking only the critical section.

3. **Static Synchronized Method:**
   Used for static methods where the lock is placed on the class object. It ensures class-level synchronization.

Programs

```
// Synchronized Method

class Counter {

  int count = 0;

  synchronized void increment() { count++; }

}

// Synchronized Block

synchronized(this) {

  count++;

}

// Static Synchronized Method

static synchronized void display() {

  System.out.println("Static synchronized");

}
```

7. Explain Inter-Thread Communication and wait(), notify(), notifyAll().

Inter-thread communication allows threads to coordinate and communicate with each other. It is mainly used when one thread depends on another, such as producer-consumer problems

Purpose of Inter-Thread Communication

1. **Avoids Polling:**
   Polling wastes CPU time by repeatedly checking conditions. Inter-thread communication avoids this inefficiency.
2. **Coordinates Threads:**
   Threads can pause and resume execution based on conditions. This improves performance and correctness.

Methods Used

1. **wait():**
   It causes the current thread to release the lock and wait. The thread resumes only when notified.
2. **notify():**
   It wakes up one waiting thread. The choice of thread is decided by JVM.
3. **notifyAll():**
   It wakes up all waiting threads. One thread gets the lock and proceeds.

Example Program

```
class Data {
 synchronized void show() throws InterruptedException {
  wait();
  System.out.println("Notified thread running");
 }
 synchronized void notifyThread() {
  notify();
 }
}
```

8. Program to Create Two Threads: One Using Thread Class and One Using Runnable.

Java allows thread creation using **Thread class** and **Runnable interface**. Both approaches execute code concurrently but differ in design flexibility

Thread Using Thread Class

1. A class extends Thread and overrides run() method.
2. start() method begins execution.

```
class MyThread extends Thread {
 public void run() {
  System.out.println("Thread class thread");
 }
}
```

Thread Using Runnable Interface

1. A class implements Runnable and defines run().
2. Runnable object is passed to Thread constructor.

```
class MyRun implements Runnable {
 public void run() {
  System.out.println("Runnable thread");
 }
}
```

Main Program

```
public class Demo {
 public static void main(String[] args) {
  new MyThread().start();
  new Thread(new MyRun()).start();
 }
}
```

9. Important Classes of Java I/O Class Hierarchy (Explain with Diagram).

Java I/O is based on **stream-oriented hierarchy** that handles input and output operations. All I/O classes are present in the java.io package

Main Abstract Classes

1. **InputStream:**
   Used for reading byte data. It is the base class for all byte input streams.

2. **OutputStream:**
   Used for writing byte data. All byte output streams extend this class.

3. **Reader:**
   Used for reading character data. It supports Unicode characters.

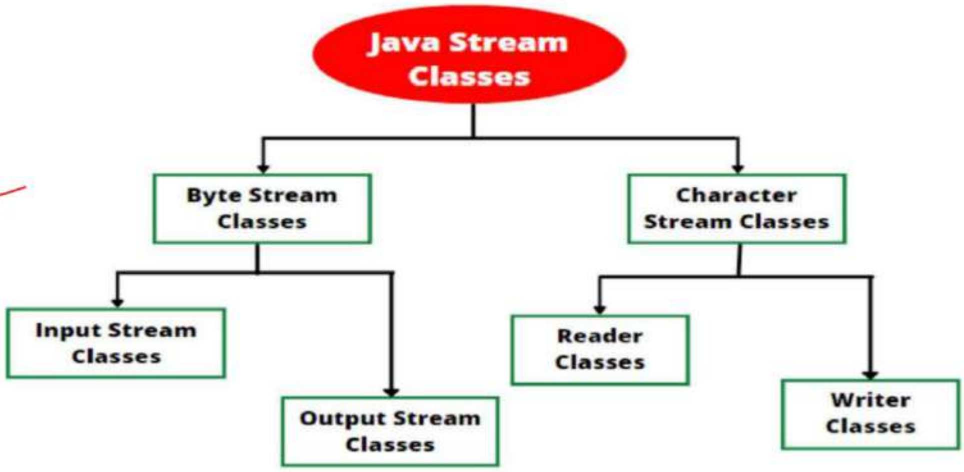
4. **Writer:**
   Used for writing character data. It handles text output.

Important Subclasses

1. **FileInputStream / FileOutputStream:**
   Used for reading and writing raw byte data from files.

2. **FileReader / FileWriter:**
   Used for reading and writing characters from files.

3. **BufferedReader / BufferedWriter:**
   Improve performance by using buffers



10. Explain Streams in Java. Differentiate Byte Stream and Character Stream.

A **stream** is a flow of data between a source and a destination. Java uses streams to perform input and output operations efficiently

Concept of Streams

1. **Input Stream:** Used to read data from a source such as keyboard or file. Data flows into the program.
2. **Output Stream:** Used to write data to a destination such as console or file. Data flows out of the program.

Byte Stream

1. **Data Type:** Works with raw binary data (8-bit bytes). Suitable for images, audio, and video files.
2. **Classes:** InputStream, OutputStream, FileInputStream, FileOutputStream.

Character Stream

1. **Data Type:** Works with 16-bit Unicode characters. Suitable for text files.
2. **Classes:** Reader, Writer, FileReader, FileWriter.

Difference Summary

1. Byte stream handles binary data, character stream handles text data.
2. Character stream automatically converts bytes to characters.

11. Explain File class, its purpose and methods. Write a program to demonstrate File class methods.

The **File class** in Java is used to represent the path of a file or directory in the file system. It does not read or write data but provides information and operations related to files and directories

Purpose of File Class

1. **File Representation:**
   The File class represents files and directories as objects. It allows Java programs to interact with the file system.
2. **File Management:**
   It is used to create, delete, rename, and check properties of files or folders. Actual data handling is done by stream classes.

Important Methods

1. **exists():**
   Checks whether a file or directory exists. It returns true if the path is valid.
2. **getName():**
   Returns the name of the file or directory. It does not return the full path.

3. **getAbsolutePath():**
   Returns the complete path of the file. It is useful for identifying file location.
4. **isFile() / isDirectory():**
   These methods check whether the object refers to a file or a directory.
5. **createNewFile():**
   Creates a new empty file if it does not exist.

Program

```
import java.io.File;
import java.io.IOException;
public class FileDemo {
 public static void main(String[] args) throws IOException {
  File f = new File("demo.txt");
  System.out.println(f.exists());
  f.createNewFile();
  System.out.println(f.getName());
  System.out.println(f.getAbsolutePath());
  System.out.println(f.isFile());
 } }
```

12. Purpose of BufferedReader and BufferedWriter. Program to Read from User and Write to File.

BufferedReader and **BufferedWriter** improve I/O performance by using a buffer. They reduce the number of direct accesses to the disk, making reading and writing faster

Purpose of BufferedReader

1. **Efficient Reading:**
   BufferedReader reads data in chunks instead of one character at a time. This improves performance.
2. **Line-by-Line Reading:**
   It provides the readLine() method to read text line by line.

Purpose of BufferedWriter

1. **Efficient Writing:**
   BufferedWriter stores data in a buffer before writing it to the file. This reduces disk I/O.
2. **newLine() Method:**
   It allows writing platform-independent line separators.

Program

```
import java.io.*;


public class BufferDemo {
 public static void main(String[] args) throws IOException {
  BufferedReader br =  new BufferedReader(new InputStreamReader(System.in));
  BufferedWriter bw =  new BufferedWriter(new FileWriter("output.txt"));
  System.out.println("Enter text:");
  String data = br.readLine();
  bw.write(data);
  bw.newLine();
  bw.close();
 }}
```

13. Explain PrintWriter Class, Its Purpose and Methods with Example.

The **PrintWriter class** is a character stream used to write formatted text data to files or console. It provides convenient methods like print(), println(), and printf()

Purpose of PrintWriter

1. **Formatted Output:**
   PrintWriter converts primitive data types into text format. This makes output human-readable.
2. **Ease of Use:**
   It does not force handling IOException in most methods, simplifying code.

Important Methods

1. **print():**
   Writes data without moving to the next line. It is useful for continuous output.
2. **println():**
   Writes data and moves the cursor to the next line. It is commonly used.
3. **printf():**
   Writes formatted output using format specifiers. It is useful for formatted numbers.
4. **close():**
   Closes the stream and releases resources.

Example Program

```
import java.io.PrintWriter;

public class PrintWriterDemo {

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

  PrintWriter pw = new PrintWriter("data.txt");

  pw.println("Hello World");

  pw.println(100);

  pw.printf("Value: %.2f", 45.678);

  pw.close();

 }}
```

14. Program to Read from User and Write to File Using Character Stream Classes.

Character stream classes are used to read and write **Unicode characters**. They are suitable for handling text files in Java

Concept

1. **FileReader:** FileReader is used to read characters from a file or input source. It works with 16-bit Unicode characters.
2. **FileWriter:** FileWriter is used to write characters to a file. It supports character-based output.

Program

```
import java.io.*;

public class CharStreamDemo {

 public static void main(String[] args) throws IOException {

  BufferedReader br =

   new BufferedReader(new InputStreamReader(System.in));

  FileWriter fw = new FileWriter("charfile.txt");


  System.out.println("Enter text:");

  String text = br.readLine();

  fw.write(text);

  fw.close();

 }}
```