1. What is Multithreading?

□ Definition:

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum CPU utilization. Each part of such a program is called a **thread**, and each thread defines a separate path of execution.

□ Why Use Multithreading?

- Efficient CPU utilization
- Better performance for multi-core systems
- Enables background tasks like file saving, printing, or animation while keeping the app responsive

□ Real-Life Examples:

- Web browsers: loading multiple tabs
- Download managers: multiple files downloading simultaneously
- Online games: background music, game logic, network communication all at once

---

2. Thread Class Introduction

Java provides the `Thread` class in the `java.lang` package to create and control threads.

□ Important Methods in Thread class:

Method	Description
`start()`	Starts a new thread
`run()`	Defines code that runs in a thread
`sleep(ms)`	Makes thread pause for a given time
`join()`	Waits for a thread to die
`isAlive()`	Checks if a thread is still running
`getName()`	Gets the name of the thread
`setName(String)`	Sets the name of the thread
`setPriority(int)`	Sets thread priority (1 to 10)

---

3. Creating Threads

□ There are two ways to create threads in Java:

A. By Extending the Thread class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running: " +
Thread.currentThread().getName());
    }
}

public class TestThread {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setName("First Thread");
        t1.start();
    }
}
```

B. By Implementing Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running using Runnable: " +
Thread.currentThread().getName());
    }
}

public class TestRunnable {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.setName("Runnable Thread");
        t1.start();
    }
}
```

□ Recommended way: **Using `Runnable`** because it allows multiple inheritance (class + interface).

---

4. Controlling Threads

Java provides methods to control thread execution:

□ A. `sleep()` method

```
class SleepExample extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println("Count: " + i);
            try {
                Thread.sleep(1000); // Sleep for 1 second
            } catch(InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}
```

□ B. `join()` method

```
class JoinExample extends Thread {
    public void run() {
        for(int i = 1; i <= 3; i++) {
            System.out.println(Thread.currentThread().getName() + ": " + i);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        JoinExample t1 = new JoinExample();
        JoinExample t2 = new JoinExample();

        t1.start();
        t1.join();  // main thread waits for t1 to finish
        t2.start();
    }
}
```

□ C. `isAlive()` method

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

    public static void main(String[] args) {
        AliveExample t = new AliveExample();
        System.out.println("Is Alive before start: " + t.isAlive());
        t.start();
        System.out.println("Is Alive after start: " + t.isAlive());
    }
}
```

---

5. Thread Synchronization

□ What is it?

When multiple threads try to access a shared resource (e.g., a variable or file), it may cause **data inconsistency**. To avoid this, Java uses synchronization.

□ Synchronized Method Example

```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}

public class SyncExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

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

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

        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println("Final Count: " + counter.count); // Should be
2000
    }
}
```

□ Synchronized Block Example

```
synchronized(counter) {
    counter.increment();
}
```

6. Inter-thread Communication

□ What is it?

It refers to the exchange of data between threads to coordinate actions. It avoids busy-waiting using `wait()`, `notify()`, and `notifyAll()` methods.

□ Important Methods (from Object class)

Method	Description
`wait()`	Causes the current thread to wait
`notify()`	Wakes up a single waiting thread
`notifyAll()`	Wakes up all waiting threads

□ Producer-Consumer Example using wait/notify

```
class Shared {
    int data;
    boolean isProduced = false;

    synchronized void produce(int value) {
        try {
            while (isProduced) wait();
            data = value;
            System.out.println("Produced: " + data);
            isProduced = true;
            notify();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }

    synchronized void consume() {
        try {
            while (!isProduced) wait();
            System.out.println("Consumed: " + data);
            isProduced = false;
            notify();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

```
public class InterThreadComm {
    public static void main(String[] args) {
        Shared shared = new Shared();

        Thread producer = new Thread(() -> {
            for(int i = 1; i <= 5; i++) shared.produce(i);
        });

        Thread consumer = new Thread(() -> {
            for(int i = 1; i <= 5; i++) shared.consume();
        });

        producer.start();
        consumer.start();
    }
}
```