SILVER OAK UNIVERSITY
EDUCATION TO INNOVATION

Course Code : 3040243201
Course Name: Java Programming
Unit No: 4
Unit Name: Multithreading Programming

SEMESTER: 1

PREPARED BY: Dr. Nikunj Raval

Multithreading is a powerful feature in Java that allows concurrent execution of two or more threads. Here's a comprehensive overview of multithreading programming in Java, covering key concepts and best practices.

- **1. The Java Thread Model**
- In Java, a thread is a lightweight process. The Java Virtual Machine (JVM) supports multithreading, enabling multiple threads to run concurrently within a single program. The Java thread model allows you to manage threads efficiently and provides various mechanisms for thread synchronization and communication.

2. Understanding Threads

A thread is an independent path of execution within a program. Each thread has its own call stack, local variables, and execution context, but they share the same memory space of the application.

3. The Main Thread

When a Java program starts, the JVM creates a main thread, which is responsible for executing the main() method. You can create additional threads to perform tasks concurrently.

4. Creating a Thread

There are two primary ways to create a thread in Java:

a. By Extending the Thread Class

You can create a new thread by extending the Thread class and overriding its run() method.

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

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Starts the thread
    }
}
```

b. **By Implementing the Runnable Interface**

Another way to create a thread is by implementing the Runnable interface.

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

public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start(); // Starts the thread
    }
}
```

5. Creating Multiple Threads

You can create multiple threads by instantiating multiple Thread or Runnable objects.

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            Thread thread = new Thread(new MyRunnable());
            thread.start();
        }
    }
}
```

6. Thread Priorities

Java threads have priorities that can influence their scheduling. You can set the priority of a thread using setPriority(), with values ranging from Thread.MIN_PRIORITY (1) to Thread.MAX_PRIORITY (10). The default priority is Thread.NORM_PRIORITY (5).

```
Thread thread = new Thread(new MyRunnable());
thread.setPriority(Thread.MAX_PRIORITY);
```

7. Synchronization

When multiple threads access shared resources, synchronization is essential to prevent data inconsistency. You can use the synchronized keyword to control access to methods or blocks of code.

- Example of Method Synchronization:

```
class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
}
```

- Example of Block Synchronization:

```
public void someMethod() {
    synchronized (this) {
        // synchronized block of code
} }
```

8. Inter-thread Communication

Threads can communicate with each other using methods such as wait(), notify(), and notifyAll(). These methods are used for coordinating actions between threads.

Example:

```
class SharedResource {
    public synchronized void producer() throws InterruptedException {
        // Produce resource
        notify(); // Notify waiting threads
    }

    public synchronized void consumer() throws InterruptedException {
        wait(); // Wait until notified
        // Consume resource
    }
}
```

- **9. Deadlocks**
- A deadlock occurs when two or more threads are waiting for each other to release resources, causing them to be stuck indefinitely. To avoid deadlocks:
- **Lock Ordering**: Always acquire locks in a consistent order.
- **Timeouts**: Use timeouts when trying to acquire locks.
- **Avoid Nested Locks**: Minimize the use of nested locks whenever possible.