Tutorial No. 6

Title: Thread Management System in Java

Batch: SY-IT(B3) **Roll No.: 16010423076** **Tutorial No.:6**

Aim: To understand and explore Thread handling mechanism in Java with the following tasks:

1. Understand the Thread Life Cycle and monitor/print the different states of a thread as it moves through its lifecycle.
2. Develop a Java application that concurrently computes prime numbers and the Fibonacci series using two separate threads. The program should efficiently utilize the CPU by performing these two independent computations in parallel.

Resources needed: Java

Theory:

Multithreading is a powerful feature in Java that allows concurrent execution of two or more parts of a program to maximize the utilization of CPU resources. It's particularly useful when you need to perform multiple tasks simultaneously, such as downloading files, processing data, or handling user interactions in a responsive application.

Multi-Threading in Java

Multithreading in Java is the ability of a CPU, or a single core in a multi-core processor, to execute multiple threads concurrently. A thread is the smallest unit of a process that can be scheduled and executed. In a multithreaded environment, each thread runs parallel to each other.

Thread Life Cycle

A thread in Java goes through various states in its life cycle:

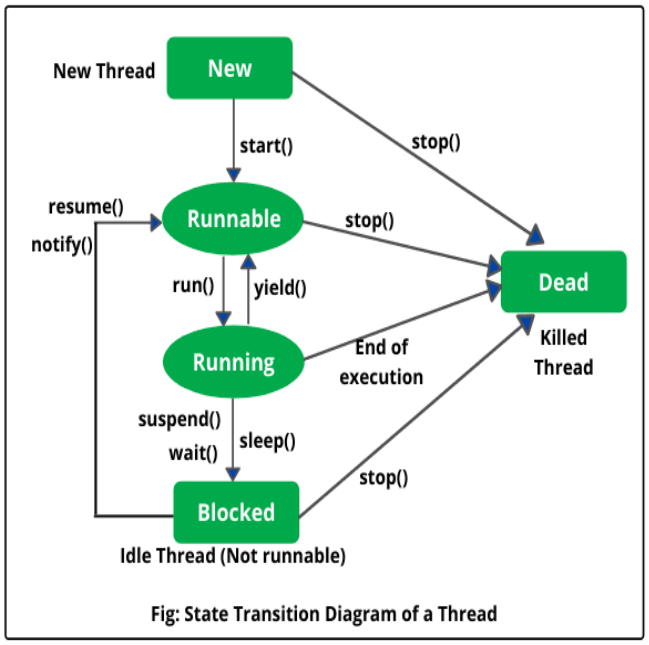

Fig: State Transition Diagram of a Thread

New: A thread is in the "New" state if you create an instance of the Thread class but before the start() method is invoked.

Runnable: When the thread's start() method is called, it enters the "Runnable" state. The thread is ready to run and waiting for CPU time.

Blocked/Waiting: The thread enters this state if it is waiting for a resource or another thread to complete its task.

Timed Waiting: This state is similar to the waiting state but with a specified time limit, after which the thread will return to the runnable state.

Terminated(Dead State): Once the thread has completed its execution or has been explicitly terminated, it enters the "Terminated" state.

Mechanism of Working of Threads:

The diagram below illustrates the life cycle of a thread in Java, showing how a thread transitions between different states during its execution. Here's a detailed explanation of each part of the diagram:

1. New State, When a thread is created, it is in the "New" state. This means that the thread has been instantiated but not yet started. In transition state, where the thread enters the "Runnable" state when the start() method is invoked.
2. Runnable State, where a thread in the "Runnable" state is ready to run and waiting for CPU time. However, it might not be executing immediately depending on the CPU's availability. Either of the state should reach after the state:
    - **To Running**: When the thread scheduler selects this thread for execution, it transitions to the "Running" state.

- **To Blocked**: If the thread needs to acquire a lock but the lock is currently held by another thread, it transitions to the "Blocked" state.
- **To Waiting**: If the thread calls methods like wait(), join(), or park(), it transitions to the "Waiting" state.
- **To Timed Waiting**: If the thread calls methods like sleep(long), wait(long), or join(long), it transitions to the "Timed Waiting" state.

3. In Running State, the thread is actively executing its code.
   a. **To Runnable**: The thread can go back to the "Runnable" state if it's preempted by the scheduler or if its time slice expires.
   b. **To Terminated**: Once the thread finishes executing its run() method, it moves to the "Terminated" state.
4. Blocked State :
   A thread enters the "Blocked" state when it attempts to acquire a lock that is held by another thread. In this state, the thread is waiting to enter a synchronized block or method.
   Once the thread successfully acquires the lock, it moves back to the "Runnable" state.
5. Waiting State:
   The thread is in the "Waiting" state when it is waiting indefinitely for another thread to perform a particular action. This can happen when a thread is waiting for a notification (wait()), another thread to complete (join() without a timeout), or a condition to be met (park()). The thread transitions back to the "Runnable" state when it receives a notification or the condition it's waiting for is met.
6. Timed Waiting State:
   The thread is in the "Timed Waiting" state when it is waiting for another thread to perform an action but only for a specified amount of time. Examples include sleep(long), wait(long), and join(long).
   After the specified time has elapsed or the thread receives the expected notification, it transitions back to the "Runnable" state.
7. Terminated/Dead State : Once a thread has completed its execution, it enters the "Terminated" state. This state indicates that the thread has finished running either because the run() method has exited or because an exception occurred that was not caught. There are no transitions out of the "Terminated" state as the thread's life cycle ends here.

Steps for Creating a Thread Using Different Classes and Interfaces:

Define the Interface (Optional):

If you want to enforce a common contract for thread-related behavior across different classes, create an interface.

Define methods that the classes implementing the interface must implement.

Example:

```
class MyRunnable implements Runnable {
  @Override
  public void run() {
    // Code that will be executed by the thread
    System.out.println("Thread is running using Runnable Interface.");
  }
```

```
}
```

Create the Implementing Class:

Create a class that implements the interface (if defined) or extends the Thread class.
Override the run() method to define the thread's behavior.
Example:

```
public class MyThread implements RunnableThread {
   @Override
   public void run() {
      // Thread's logic here
      System.out.println("Thread is running...");
   }
}
```

Create a Thread Object:

Instantiate a Thread object, passing the implementing class as a constructor argument.
Example:

```
Thread thread = new Thread(new MyThread());
```

Start the Thread:

Call the start() method on the Thread object to begin its execution.
Example:

```
thread.start();
```

Start with Runnable Interface:

```
public class Main {

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

Task to be performed in Tutorial:

Problem Statement: Concurrent Computation of Prime Numbers and Fibonacci Series

A. Prime Number Computation:

One thread should be responsible for calculating the first N prime numbers.
The prime number calculation should be done by checking each number starting from 2 for primality until N primes have been found. The thread should print each prime number as it is found.

B. Fibonacci Series Computation:
A second thread should compute the first M numbers in the Fibonacci series.

The Fibonacci series is defined as follows: F(0) = 0, F(1) = 1, and F(n) = F(n-1) + F(n-2) for n >= 2.

The thread should print each Fibonacci number as it is computed.

C. Concurrency:
The program should run both threads concurrently, allowing the prime number calculation and Fibonacci series computation to proceed independently of each other. The threads should start their computations at the same time, and both should continue until their respective tasks are complete.

D. Thread Management:
After starting both threads, the main thread should wait for both computations to finish before terminating the program.
The program should handle any potential thread interruptions gracefully.

User Input:

The program should prompt the user to input the values of N (for prime numbers) and M (for Fibonacci numbers) at the start.

Expected Output:
The program should concurrently print the first N prime numbers and the first M Fibonacci numbers.
The output should demonstrate that the two computations are running in parallel, with prime numbers and Fibonacci numbers being printed interleaved based on the scheduling of the threads by the JVM.

Program :

```
class PrimeNumberThread extends Thread {
   private int n;

   PrimeNumberThread(int n) {
      this.n = n;
   }

   @Override
   public void run() {
      int count = 0;
      int num = 2;  // Start checking for primes from 2
      while (count < n) {
         if (isPrime(num)) {
            System.out.println("Prime: " + num);
            count++;
         }
         num++;
      }
```

```
    }

    private boolean isPrime(int num) {
        if (num <= 1) {
            return false;
        }
        for (int i = 2; i <= Math.sqrt(num); i++) {
            if (num % i == 0) {
                return false;
            }
        }
        return true;
    }
}

class FibonacciThread extends Thread {
    private int m;

    FibonacciThread(int m) {
        this.m = m;
    }

    @Override
    public void run() {
        int a = 0, b = 1;
        for (int i = 0; i < m; i++) {
            if (i == 0) {
                System.out.println("Fibonacci: " + a);
            } else if (i == 1) {
                System.out.println("Fibonacci: " + b);
            } else {
                int next = a + b;
                System.out.println("Fibonacci: " + next);
                a = b;
                b = next;
            }
        }
    }
}

public class ConcurrentComputation {
    public static void main(String[] args) {
        int n = 10;  // Number of primes
        int m = 10;  // Number of Fibonacci numbers
```

```
        PrimeNumberThread primeThread = new PrimeNumberThread(n);
        FibonacciThread fibonacciThread = new FibonacciThread(m);

        primeThread.start();
        fibonacciThread.start();

        try {
            primeThread.join();  // Wait for primeThread to finish
            fibonacciThread.join();  // Wait for fibonacciThread to finish
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Both computations are done!");
    }
}
```

Output :

Output

```
java -cp /tmp/CNAN6CEpET/ConcurrentComputation
Prime: 2
Prime: 3
Prime: 5
Prime: 7
Fibonacci: 0
Prime: 11
Prime: 13
Prime: 17
Prime: 19
Prime: 23
Prime: 29
Fibonacci: 1
Fibonacci: 1
Fibonacci: 2
Fibonacci: 3
Fibonacci: 5
Fibonacci: 8
Fibonacci: 13
Fibonacci: 21
Fibonacci: 34
Both computations are done!

=== Code Execution Successful ===
```

Post Lab Questions:-

1. What is the used of Synchronized keyword in Java?

The `synchronized` keyword in Java is used to control access to a block of code or a method by allowing only one thread to execute it at a time. This is helpful when you have multiple threads trying to change shared data. By using `synchronized`, you prevent two or more threads from modifying that data at the same time, which could lead to errors. In simple terms, it ensures that only one user or thread can perform an action at a given moment, keeping things safe and organized.

2. Develop a Java-based ticket booking system where multiple users (represented by threads) attempt to book tickets concurrently. The system should handle race conditions, ensuring that no more tickets are booked than are available, and should provide feedback to users based on the success or failure of their booking attempts.

```
import java.util.Scanner;

public class simpleTicketBooking {
   private static int availableTickets = 10;

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);

      while (availableTickets>0) {
      System.out.println("Available tickets : " + availableTickets);
      System.out.print("Enter number of tickets to book (or 0 to exit): ");
      int ticketsToBook = scanner.nextInt();
      if (ticketsToBook == 0) {
      break;
      }

      if (ticketsToBook <= availableTickets) {
      availableTickets -= ticketsToBook;
      System.out.println("Booked " + ticketsToBook + " ticket(s) successfully!");
      }
      else {
      System.out.println("Sorry, not enough tickets available.");
      }
      }

      System.out.println("Booking ended. Remaining tickets: " + availableTickets);
```

```
        scanner.close();
    }
}
```

Output

```
java -cp /tmp/cf9XW5QhsF/simpleTicketBooking
Available tickets : 10
Enter number of tickets to book (or 0 to exit): 1
Booked 1 ticket(s) successfully!
Available tickets : 9
Enter number of tickets to book (or 0 to exit): 15
Sorry, not enough tickets available.
Available tickets : 9
Enter number of tickets to book (or 0 to exit): 9
Booked 9 ticket(s) successfully!
Booking ended. Remaining tickets: 0

=== Code Execution Successful ===
```

Outcomes:

CO3: Demonstrate the concept of packages, multithreading and exception handling in java

---

Conclusion: (Conclusion to be based on the outcomes achieved)

I learned how to create and manage threads in Java using classes that extend the Thread class. I now understand how to run multiple threads concurrently, like generating prime numbers and Fibonacci series at the same time. I also learned about the importance of the synchronized keyword in preventing race conditions when multiple threads share resources. Finally, the ticket booking system example showed me how to handle concurrent users and ensure correct ticket counts in a multi-threaded environment.

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of faculty in-charge with date

References Books

1. Herbert Schildt; JAVA The Complete Reference; Seventh Edition, Tata McGraw- Hill Publishing Company Limited 2007.
2. Java 7 Programming - Black Book : Kogent Learning Solutions Inc.
3. Sachin Malhotra, Saurabh Chaudhary “Programming in Java”, Oxford University Press, 2010
4. Jaime Nino, Frederick A. Hosch, ‘An introduction to Programming and Object Oriented Design using Java’, Wiley Student Edition.