SENJUTI GHOSAL
RA2111030010096
COGNIZANT WEEK2

Exercise 1: Understanding Thread States

Objectives:

- Learn about different states of a thread.
- Understand how threads transition between states.

Business Scenario:

You are developing a logging system that needs to monitor the states of various threads in a multithreaded application. You need to create a thread and observe its state transitions.

Tasks:

1. **Create a New Java Class:**
    - Create a Java class named **ThreadStateLogger**.
2. **Implement Thread States Logging:**
    - Define a new thread class that overrides the **run** method to perform a simple task (e.g., printing numbers).
    - In the **ThreadStateLogger** class, create an instance of this thread.
    - Log the state of the thread at various points: before starting, after starting, during execution, and after completion.
3. **Execute the Program:**
    - Run the **ThreadStateLogger** class and observe the output showing the state transitions of the thread.

```
State before starting the thread: NEW
State after starting the thread: RUNNABLE
Current state: RUNNABLE
Running... 0
TING
Current state: TIMED_WAITING
Running... 3
Current state: TIMED_WAITING
Current state: TIMED_WAITING
Running... 4
Current state: TIMED_WAITING
Current state: TIMED_WAITING
State after completion: TERMINATED
```

ThreadStateLogger:

```
public class ThreadStateLogger {
    public static void main(String[] args) {
        Thread thread = new Thread(new SimpleTask());

        // Log state before starting the thread
        System.out.println("State before starting the thread: " + thread.getState());

        // Start the thread
        thread.start();

        // Log state after starting the thread
        System.out.println("State after starting the thread: " + thread.getState());

        // Monitor state during execution
        while (thread.getState() != Thread.State.TERMINATED) {
            System.out.println("Current state: " + thread.getState());
            try {
                Thread.sleep(500); // Sleep for a while to observe state changes
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Log state after completion
        System.out.println("State after completion: " + thread.getState());
    }
}
```

```
class SimpleTask implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Running... " + i);
            try {
                Thread.sleep(1000); // Simulate some work
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Exercise 2: Creating and Running Threads

Objectives:

- Create and start threads in Java.
- Understand the main thread and how other threads interact with it.

Business Scenario:

You are developing a simulation where multiple sensors collect data simultaneously. Each sensor should run in its own thread.

Tasks:

1. **Create a New Java Class:**
    - Create a Java class named **SensorSimulation**.
2. **Define Sensor Threads:**
    - Create a thread class named **Sensor** that simulates data collection by printing random data at regular intervals.
    - In the **SensorSimulation** class, create and start multiple **Sensor** threads.
3. **Main Thread Interaction:**
    - In the **SensorSimulation** class, ensure the main thread waits for all sensor threads to complete before exiting.
    - Use **join** method to make the main thread wait for the sensor threads.

4. **Execute the Program:**

    - Run the **SensorSimulation** class and observe the concurrent execution of sensor threads.

```
ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\SENJUTI\AppData\Roaming\Code\User\workspaceStorage\72a5cd4a
be3360c7f7cbaabad7406608\redhat.java\jdt_ws\cognizant_3b617e60\bin' 'Thread.SensorSimulation'
Sensor2 collected data: 81
Sensor1 collected data: 11
Sensor3 collected data: 99
Sensor2 collected data: 35
Sensor2 collected data: 35
Sensor3 collected data: 55
Sensor1 collected data: 61
Sensor1 collected data: 82
Sensor3 collected data: 33
Sensor2 collected data: 32
Sensor1 collected data: 39
Sensor3 collected data: 35
Sensor2 collected data: 36
Sensor1 collected data: 21
Sensor3 collected data: 17
Sensor2 collected data: 18
All sensor threads have completed.
```

SensorSimulation:

```
package Thread;

import java.util.Random;


public class SensorSimulation {

   public static void main(String[] args) {

      Thread sensor1 = new Thread(new Sensor("Sensor1"));

      Thread sensor2 = new Thread(new Sensor("Sensor2"));

      Thread sensor3 = new Thread(new Sensor("Sensor3"));


      sensor1.start();

      sensor2.start();

      sensor3.start();


      try {

         sensor1.join();

         sensor2.join();

         sensor3.join();
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("All sensor threads have completed.");
    }
}
class Sensor implements Runnable {
    private String name;
    private Random random = new Random();
    public Sensor(String name) {
        this.name = name;
    }
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(name + " collected data: " + random.nextInt(100));
            try {
                Thread.sleep(1000); // Simulate data collection interval
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Exercise 3: Using Sleep, Yield, and Join

Objectives:

- Understand the use of **sleep**, **yield**, and **join** methods in thread management.

Business Scenario:

You are developing a task scheduler where tasks need to be paused, yielded to other tasks, or waited upon.

Tasks:

1. **Create a New Java Class:**
    - Create a Java class named **TaskScheduler**.
2. **Define Task Threads:**
    - Create a thread class named **Task** that performs a sequence of operations.
    - Use **sleep** to pause the task for a few seconds.
    - Use **yield** to allow other tasks to execute.
    - Use **join** to wait for a dependent task to complete.
3. **Implement Task Scheduling:**
    - In the **TaskScheduler** class, create and start multiple Task threads.
    - Demonstrate the use of **sleep**, **yield**, and **join** in the task threads.
4. **Execute the Program:**
    - Run the **TaskScheduler** class and observe the task scheduling behavior.

```
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>  & 'C:\Program Files\Java\jdk-17\bin\java.exe' '-XX:+
f7cbaabad7406608\redhat.java\jdt_ws\cognizant_3b617e60\bin' 'Thread.TaskScheduler'
Task3 is running...
Task1 is running...
Task1 is running...
Task3 is running...
Task1 is running...
Task3 is running...
Task2 is running...
Task2 is running...
Task2 is running...
All tasks have completed.
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>
```

TaskScheduler.java:

```
package Thread;

public class TaskScheduler {

    public static void main(String[] args) {

        Task task1 = new Task("Task1");
```

```
        Task task2 = new Task("Task2", task1);
        Task task3 = new Task("Task3");

        task1.start();
        task2.start();
        task3.start();

        try {
            task1.join();
            task2.join();
            task3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("All tasks have completed.");
    }
}

class Task extends Thread {
    private String name;
    private Thread dependency;

    public Task(String name) {
        this.name = name;
    }
```

```
    public Task(String name, Thread dependency) {
        this.name = name;
        this.dependency = dependency;
    }

    @Override
    public void run() {
        if (dependency != null) {
            try {
                dependency.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        for (int i = 0; i < 3; i++) {
            System.out.println(name + " is running...");
            if (i == 1) {
                Thread.yield(); // Yield to other threads
            }
            try {
                Thread.sleep(1000); // Pause for a while
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Exercise 4: Synchronization

Objectives:

- Understand synchronization in multithreaded applications.
- Avoid data inconsistency with synchronized methods and blocks.

Business Scenario:

You are developing a banking system where multiple threads perform transactions on the same account. Ensure that transactions are synchronized to avoid data inconsistency.

Tasks:

1. **Create a New Java Class:**
    - Create a Java class named **BankAccount**.
2. **Define Account Operations:**
    - Create methods for deposit and withdrawal operations.
    - Use synchronized methods or blocks to ensure thread safety.
3. **Create Transaction Threads:**
    - Create a thread class named **Transaction** that performs deposit and withdrawal operations on a shared **BankAccount** instance.
4. **Implement and Test Synchronization:**
    - In the **BankAccount** class, create and start multiple **Transaction** threads.
    - Observe and ensure that the balance updates correctly with synchronized operations.
5. **Execute the Program:**
    - Run the **BankAccount** class and test the synchronized transaction operations.

```
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>  & 'C:\Program Files\Java\jdk-17\bin\java.exe' '
-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\SENJUTI\AppData\Roaming\Code\User\workspaceStorag
e\72a5cd4abe3360c7f7cbaabad7406608\redhat.java\jdt_ws\cognizant_3b617e60\bin' 'Thread.BankAccount'
Thread-0 deposited: 1000.0, Current balance: 1000.0
Thread-1 withdrew: 500.0, Current balance: 500.0
Thread-2 withdrew: 200.0, Current balance: 300.0
Final balance: 300.0
PS C:\Users\SENJUTI\OneDrive\Documents\Desktop\cognizant>
```

BankAccount.java:

```
public class BankAccount {
    private double balance;

    public synchronized void deposit(double amount) {
        balance += amount;
        System.out.println(Thread.currentThread().getName() + " deposited: " + amount + ", Current balance: " + balance);
    }

    public synchronized void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println(Thread.currentThread().getName() + " withdrew: " + amount + ", Current balance: " + balance);
        } else {
            System.out.println(Thread.currentThread().getName() + " attempted to withdraw: " + amount + ", Insufficient balance");
        }
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        Thread t1 = new Thread(new Transaction(account, 1000, "deposit"));
        Thread t2 = new Thread(new Transaction(account, 500, "withdraw"));
        Thread t3 = new Thread(new Transaction(account, 200, "withdraw"));

        t1.start();
        t2.start();
        t3.start();
```

```
        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final balance: " + account.balance);
    }
}

class Transaction implements Runnable {
    private BankAccount account;
    private double amount;
    private String type;

    public Transaction(BankAccount account, double amount, String type) {
        this.account = account;
        this.amount = amount;
        this.type = type;
    }

    @Override
    public void run() {
        if (type.equals("deposit")) {
            account.deposit(amount);
        } else if (type.equals("withdraw")) {
            account.withdraw(amount);
```

```
        }
    }
}
```

Exercise 5: Inter-Thread Communication

Objectives:

- Learn about inter-thread communication using **wait**, **notify**, and **notifyAll**.

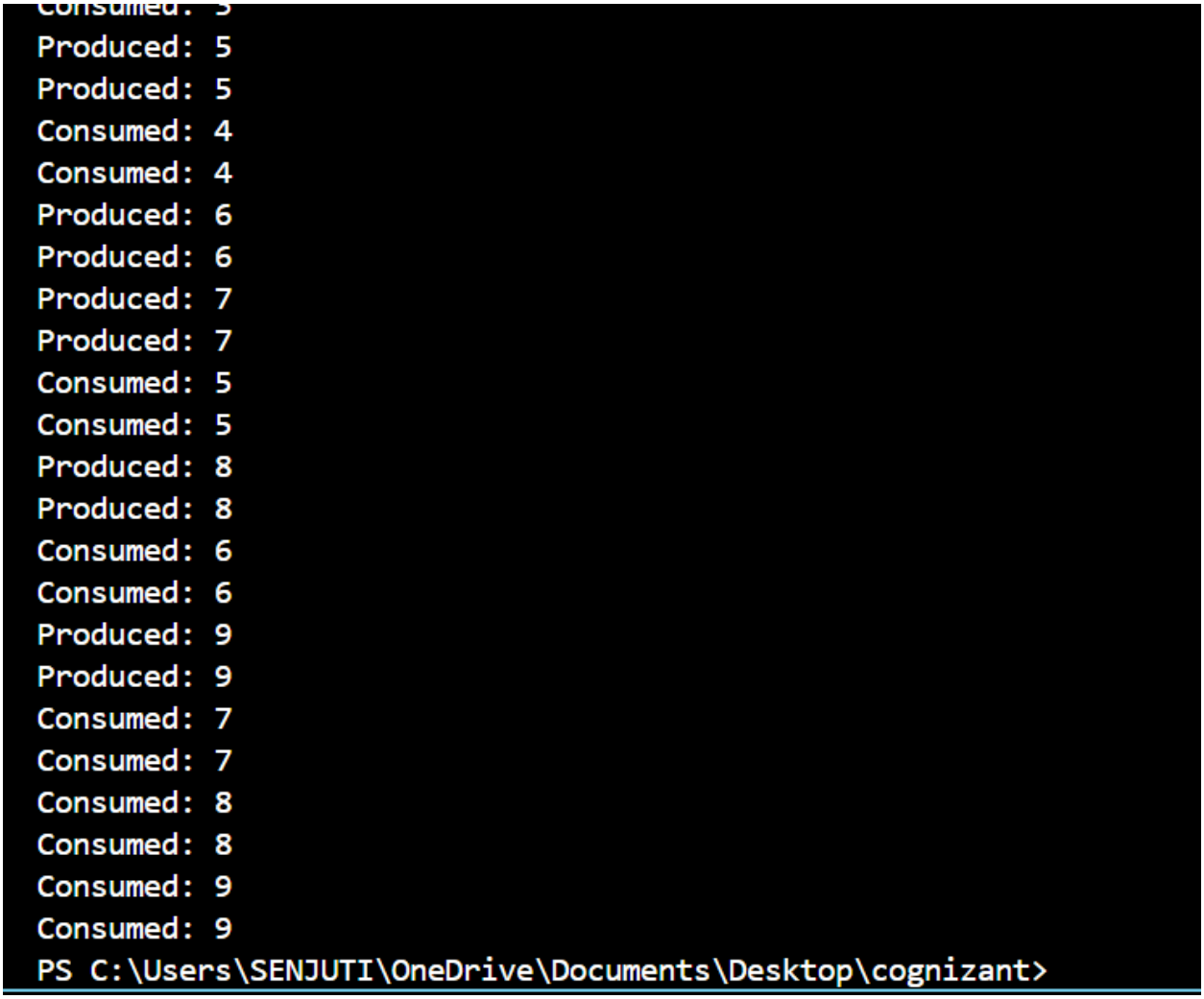
Business Scenario:

You are developing a producer-consumer system where multiple producer threads generate data and multiple consumer threads process data. Implement inter-thread communication to synchronize data production and consumption.

Tasks:

1. **Create a New Java Class:**
    - Create a Java class named **ProducerConsumer**.
2. **Define Shared Resource:**
    - Create a class named **DataQueue** that holds the data queue and methods for adding and removing data.
    - Use **wait**, **notify**, and **notifyAll** methods for inter-thread communication.
3. **Create Producer and Consumer Threads:**
    - Create thread classes named **Producer** and **Consumer** that interact with the **DataQueue** to produce and consume data.
4. **Implement and Test Inter-Thread Communication:**
    - In the **ProducerConsumer** class, create and start multiple **Producer** and **Consumer** threads.
    - Ensure proper synchronization between producers and consumers using **wait** and **notify**.
5. **Execute the Program:**
    - Run the **ProducerConsumer** class and observe the inter-thread communication.

```
ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\SENJUTI\AppData\Roaming\Code\User\workspaceStorage\72a5cd4a
be3360c7f7cbaabad7406608\redhat.java\jdt_ws\cognizant_3b617e60\bin' 'Thread.ProducerConsumer'
Produced: 0
Consumed: 0
Produced: 0
Consumed: 0
Produced: 1
Produced: 1
Consumed: 1
Consumed: 1
Produced: 2
Produced: 2
Produced: 3
Consumed: 2
Consumed: 2
Produced: 3
Produced: 4
Produced: 4
Consumed: 3
Consumed: 3
Produced: 5
Produced: 5
Consumed: 4
Consumed: 4
Produced: 6
Produced: 6
Produced: 7
Produced: 7
```



ProducerConsumer.java:

```
import java.util.LinkedList;
import java.util.Queue;
public class ProducerConsumer {
   public static void main(String[] args) {
      DataQueue dataQueue = new DataQueue();
      Thread producer1 = new Thread(new Producer(dataQueue));
      Thread producer2 = new Thread(new Producer(dataQueue));
      Thread consumer1 = new Thread(new Consumer(dataQueue));
      Thread consumer2 = new Thread(new Consumer(dataQueue));
      producer1.start();
```

```
        producer2.start();
        consumer1.start();
        consumer2.start();
        try {
            producer1.join();
            producer2.join();
            consumer1.join();
            consumer2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
class DataQueue {
    private Queue<Integer> queue = new LinkedList<>();
    private final int LIMIT = 10;
    public synchronized void addData(int data) {
        while (queue.size() == LIMIT) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        queue.add(data);
        System.out.println("Produced: " + data);
        notifyAll();
    }
    public synchronized int getData() {
```

```
        while (queue.isEmpty()) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        int data = queue.poll();
        System.out.println("Consumed: " + data);
        notifyAll();
        return data;
    }
}

class Producer implements Runnable {
    private DataQueue dataQueue;

    public Producer(DataQueue dataQueue) {
        this.dataQueue = dataQueue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            dataQueue.addData(i);
            try {
                Thread.sleep(1000); // Simulate time taken to produce data
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
        }
    }
}

class Consumer implements Runnable {
    private DataQueue dataQueue;

    public Consumer(DataQueue dataQueue) {
        this.dataQueue = dataQueue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            dataQueue.getData();
            try {
                Thread.sleep(1500); // Simulate time taken to consume data
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Exercise 6: Enhanced Multithreading and Concurrency Features

Objectives:

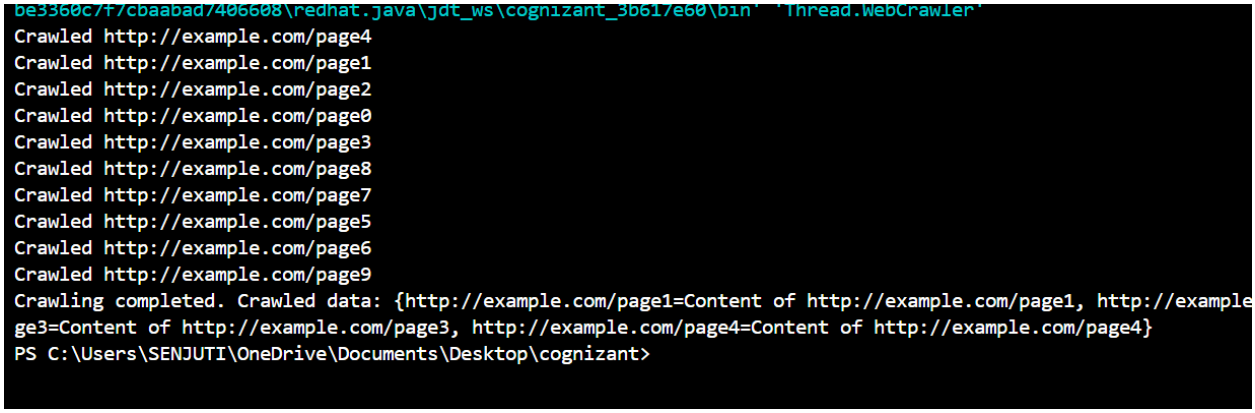
- Utilize advanced concurrency features in Java.
- Implement thread-safe collections and executors.

Business Scenario:

You are developing a web crawler that retrieves and processes web pages concurrently. Use Java's concurrency utilities to manage threads and ensure thread safety.

Tasks:

1. **Create a New Java Class:**
    - Create a Java class named **WebCrawler**.
2. **Use Thread Pools:**
    - Use **ExecutorService** to manage a pool of threads for crawling web pages.
    - Implement a thread class named **CrawlerTask** that performs the web page retrieval.
3. **Use Concurrent Collections:**
    - Use thread-safe collections like **ConcurrentHashMap** to store crawled data.
    - Ensure proper synchronization when accessing the collection.
4. **Implement and Test Enhanced Concurrency:**
    - In the **WebCrawler** class, create and start multiple **CrawlerTask** threads using **ExecutorService**.
    - Monitor and log the status of the crawling process.
5. **Execute the Program:**
    - Run the **WebCrawler** class and observe the enhanced multithreading and concurrency features in action.



WebCrawler.java:

```
import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class WebCrawler {

   public static void main(String[] args) {

      ExecutorService executor = Executors.newFixedThreadPool(5);
```

```
        ConcurrentHashMap<String, String> crawledData = new ConcurrentHashMap<>();
        for (int i = 0; i < 10; i++) {
            executor.submit(new CrawlerTask("http://example.com/page" + i, crawledData));
        }
        executor.shutdown();
        while (!executor.isTerminated()) {

        }

        System.out.println("Crawling completed. Crawled data: " + crawledData);
    }
}

class CrawlerTask implements Runnable {
    private String url;
    private ConcurrentHashMap<String, String> crawledData;

    public CrawlerTask(String url, ConcurrentHashMap<String, String> crawledData) {
        this.url = url;
        this.crawledData = crawledData;
    }

    @Override
    public void run() {
        // Simulate crawling the web page and retrieving data
        String data = "Content of " + url;
        crawledData.put(url, data);
        System.out.println("Crawled " + url);
    }
```

}