Name

Aryan Ahmad

Reg No

FA22-BSE-003

Question 1: Single-Threaded Execution

```
public class SingleThreadedExample {
   public static void main(String[] args) {
      // Simple single-threaded execution
      System.out.println("Main thread starting execution");

      // Task 1
      for (int i = 1; i <= 5; i++) {
         System.out.println("Task 1 - Count: " + i);
         try {
            Thread.sleep(500);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }

      // Task 2
      for (int i = 5; i >= 1; i--) {
         System.out.println("Task 2 - Countdown: " + i);
         try {
            Thread.sleep(500);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }

      System.out.println("Main thread completed execution");
   }
}
```

Question 2: Immutability

```
import java.util.List;
import java.util.ArrayList;

// Public class (File should be named "ImmutablePerson.java")
public final class ImmutablePerson {
   private final String name;
   private final int age;
   private final List<String> hobbies;

   public ImmutablePerson(String name, int age, List<String> hobbies) {
      this.name = name;
      this.age = age;
      this.hobbies = new ArrayList<>(hobbies);
   }

   public String getName() {
      return name;
   }

   public int getAge() {
      return age;
   }

   public List<String> getHobbies() {
      return new ArrayList<>(hobbies);
   }

   @Override
   public String toString() {
      return "ImmutablePerson{name='" + name + "', age=" + age + ", hobbies=" + hobbies +
"}";
   }
}

// Non-public class (Allowed in same file)
class ImmutableExample {
   public static void main(String[] args) {
      List<String> hobbies = new ArrayList<>();
      hobbies.add("Reading");
      hobbies.add("Hiking");

      ImmutablePerson person = new ImmutablePerson("Alice", 30, hobbies);
```

```
        System.out.println("Original: " + person);

        hobbies.add("Swimming"); // Won't affect immutable object
        System.out.println("After modifying original list: " + person);

        List<String> personHobbies = person.getHobbies();
        personHobbies.add("Painting"); // Won't affect immutable object
        System.out.println("After modifying returned list: " + person);
    }
}
```

Question 3: RPC and RMI

RPC (Using Socket Programming)

```
import java.rmi.*;
import java.rmi.server.*;

// 1. Define Remote Interface
interface Greeting extends Remote {
    String sayHello(String name) throws RemoteException;
}

// 2. Implement Remote Interface
class GreetingServer extends UnicastRemoteObject implements Greeting {
    public GreetingServer() throws RemoteException {
        super();
    }

    // Remote method implementation
    public String sayHello(String name) throws RemoteException {
        return "Hello, " + name + " from RMI Server!";
    }
}

// 3. Server Main Class
public class RMIServer {
    public static void main(String[] args) {
        try {
            // Start RMI Registry
            java.rmi.registry.LocateRegistry.createRegistry(1099);
```

```
        // Bind Remote Object
        GreetingServer server = new GreetingServer();
        Naming.rebind("GreetingService", server);

        System.out.println("RMI Server ready...");
      } catch (Exception e) {
        System.err.println("RMI Server exception: " + e.getMessage());
        e.printStackTrace();
      }
    }
}
```

RMI

```
import java.rmi.*;

public class RMIClient {
  public static void main(String[] args) {
    try {
      // Lookup Remote Object
      Greeting greeting = (Greeting) Naming.lookup("rmi://localhost/GreetingService");

      // Call Remote Method (RMI)
      String response = greeting.sayHello("Alice");
      System.out.println("RMI Response: " + response);
    } catch (Exception e) {
      System.err.println("RMI Client exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Question 4: Concurrency Patterns

1. Producer-Consumer

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class ProducerConsumerExample {
```

```
  public static void main(String[] args) {
    BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(5);

    // Producer
    Thread producer = new Thread(() -> {
      try {
        for (int i = 0; i < 10; i++) {
          queue.put(i);
          System.out.println("Produced: " + i);
          Thread.sleep(200);
        }
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    });

    // Consumer
    Thread consumer = new Thread(() -> {
      try {
        for (int i = 0; i < 10; i++) {
          Integer value = queue.take();
          System.out.println("Consumed: " + value);
          Thread.sleep(500);
        }
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    });

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

2. Actor Model

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class SimpleActor {
  private final ExecutorService executor = Executors.newSingleThreadExecutor();
```

```
  public void tell(String message) {
    executor.execute(() -> {
      System.out.println("Actor received: " + message + " on thread " +
Thread.currentThread().getName());
    });
  }

  public void shutdown() throws InterruptedException {
    executor.shutdown();
    executor.awaitTermination(1, TimeUnit.SECONDS);
  }
}

public class ActorModelExample {
  public static void main(String[] args) throws InterruptedException {
    SimpleActor actor = new SimpleActor();

    // Send messages to actor
    actor.tell("Message 1");
    actor.tell("Message 2");
    actor.tell("Message 3");

    Thread.sleep(1000); // Wait for messages to process
    actor.shutdown();
  }
}
```

3. Map-Reduce

```
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
import java.util.stream.Collectors;

public class MapReduceExample {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    // Map phase (transform each element)
    List<Integer> squaredNumbers = numbers.parallelStream()
```

```
        .map(n -> n * n)
        .collect(Collectors.toList());
    System.out.println("Squared numbers: " + squaredNumbers);

    // Reduce phase (aggregate results)
    int sum = squaredNumbers.parallelStream()
        .reduce(0, Integer::sum);
    System.out.println("Sum of squares: " + sum);

    // More complex example with ForkJoinPool
    try (ForkJoinPool pool = new ForkJoinPool()) {
        SumTask task = new SumTask(numbers);
        int result = pool.invoke(task);
        System.out.println("Sum computed with ForkJoin: " + result);
    }
  }

  static class SumTask extends RecursiveTask<Integer> {
    private final List<Integer> numbers;

    SumTask(List<Integer> numbers) {
        this.numbers = numbers;
    }

    @Override
    protected Integer compute() {
        if (numbers.size() <= 2) {
            return numbers.stream().mapToInt(Integer::intValue).sum();
        } else {
            int mid = numbers.size() / 2;
            SumTask left = new SumTask(numbers.subList(0, mid));
            SumTask right = new SumTask(numbers.subList(mid, numbers.size()));
            left.fork();
            return right.compute() + left.join();
        }
    }
  }
}
```

4. Worker Pool

```
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class WorkerPoolExample {
   public static void main(String[] args) throws InterruptedException {
      // Create a pool of 3 worker threads
      ExecutorService executor = Executors.newFixedThreadPool(3);

      // Submit 10 tasks to the pool
      for (int i = 1; i <= 10; i++) {
         final int taskId = i;
         executor.submit(() -> {
            System.out.println("Task " + taskId + " started by " +
Thread.currentThread().getName());
            try {
               Thread.sleep(1000); // Simulate work
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            System.out.println("Task " + taskId + " completed by " +
Thread.currentThread().getName());
         });
      }

      // Shutdown the pool
      executor.shutdown();
      executor.awaitTermination(1, TimeUnit.MINUTES);
      System.out.println("All tasks completed");
   }
}
```