Asynchronous Programming

1. Computing Models

Sequential Computing

- **Definition**: Computations are executed one at a time. Each task must finish before the next begins.
- **Visualization**:

```
Task1 ---> Task2 ---> Task3
```

Concurrent Computing

- **Definition**: Computations are executed during overlapping time periods, but not necessarily simultaneously.
- **Visualization**:

```
Task1 --->
   Task2 --->
      Task3 --->
```

Parallel Computing

- **Definition**: Multiple computations are executed simultaneously on separate processing elements.
- **Visualization**:

```
Task1 --->
Task2 --->
Task3 --->
```

2. Programming Models

Synchronous Programming

- **Definition**: The program waits for a task to finish before continuing.
- **Visualization**:

```
Main Program ----> [Task] ----> Main Program Continues
```

- **Example**:

```
task.run(); // Program waits here until task is complete
```

Asynchronous Programming

- **Definition**: The program initiates a task and continues without waiting for it to finish.
- **Visualization**:

```
                     Task Executes
Main Program ----------------------------------->
                              (Main Program continues)
```

- **Example**:

```
new Thread(task).start(); // Task runs independently
```

3. Threads and Multithreading

Understanding Threads

- **Thread**: An independent path of execution within a program.
    - To execute any program, your OS creates a thread.
- **Attributes of a Thread**: The thread encapsulates the following information:
    1. **Instruction Pointer**: Current execution point.
    2. **Call Stack**: Methods currently executing.

3. **Memory**: Shared heap memory.
    - Each thread has its own, **separate instruction pointer, call stack, but have shared memory.**

- **Multithreading**: Running multiple threads simultaneously within a program.
- `Thread` **Class in Java:**
    - Is built-in to Java.
    - Represents a thread of execution.
    - **Must be given a `Runnable` object in the constructor, which will then be executed using `start()` method.**
    - **Executes the `Runnable` *asynchronously (in parallel)*. ⇒ Allows for asynchronous task execution.**
    - Can just use it as is - no need to re-implement!
    - **Example**:

```
Runnable task1 = () -> {
    for (int i = 1; i <= 10; i++) {
        System.out.println(i);
    }
};
Thread thread = new Thread(task1);
thread.start();
```

Java's `Runnable` Interface

- **Definition**:

```
public interface Runnable {
    void run();
}
```

- **Purpose**: Represents a task that can be executed, either synchronously or asynchronously.

- Is built-in to Java.

Creating a Runnable Object

- **Using Lambda Expressions**: Is a single-method interface ⇒ Can use Lambda expressions.

```
Runnable task = () -> {
    // Task implementation
};
```

- **Example**:

```
Runnable printNumbers = () -> {
    for (int i = 1; i <= 10; i++) {
        System.out.println(i);
    }
};
```

4. Synchronous vs. Asynchronous Execution

Running Tasks Synchronously

- **Execution**:

```
task.run(); // Executes in the current thread
```

- **Example**:

```
System.out.println("Start");
task.run();
System.out.println("End");
```

- **Behavior**: The program waits for `task.run()` to complete before proceeding.

Running Tasks Asynchronously

- **Execution**:

```
Thread thread = new Thread(task); // Using Thread
thread.start(); // Executes in a new thread
```

- **Example**:

```
System.out.println("Start");
Thread thread = new Thread(task);
thread.start();
System.out.println("End");
```

- **Behavior**: The program does not wait for the task to complete.

5. Waiting for Threads to Finish

The `join()` Method

- **Purpose**: Makes the main thread wait for a thread to finish its execution.
- **Usage**:

```
thread.join();
```

- **Example:** Without `join()`:

```
Thread thread1 = new Thread(task1);
Thread thread2 = new Thread(task2);
thread1.start();
thread2.start();

System.out.println("Both threads have finished.");
```

  - **Behavior**: Both threads execute executes asynchronously, but the `println` statement will also be executed asynchronously.
  - **Sample Output:** (Assuming both tasks prints numbers from 0 → 10)

```
Both threads have finished.
00 1 1 2 3 2 4 3 5 4 5 6 7 8 6 9 7 8 9
```

- **Example**: With `join()`:

  ```
  Thread thread1 = new Thread(task1);
  Thread thread2 = new Thread(task2);
  thread1.start();
  thread2.start();

  thread1.join();
  thread2.join();

  System.out.println("Both threads have finished.");
  ```

  - **Behavior**: The **main** thread pauses until both `thread1` and `thread2` have completed.
  - `thread1` and `thread2` still executes asynchronously.
  - **Sample Output:** (Assuming both tasks prints numbers from 0 → 10)

    ```
    0 1 2 3 0 1 2 3 4 5 64 7 8 9 5 6 7 8 9 Finished!
    ```

5. Race Conditions

Understanding Race Conditions

- **Definition**: A situation where the program's behavior depends on the sequence or timing of uncontrollable events.
  - A segment of concurrent code where the timing of execution affects the result.
- **Cause**: Multiple threads accessing shared resources without proper synchronization.
  - Occur when two or more threads share memory.

- Multiple threads reading from or writing to the same object.

- **Implications**:
  - Unpredictable results.
  - Difficult to debug.

Example: Shared Counter Class

```
public class Counter {
    private int value = 0;

    public void increment() {
        value = value + 1;
    }

    public void decrement() {
        value = value - 1;
    }

    public int getValue() {
        return value;
    }
}
```

Usage in Multithreading

```
Counter counter = new Counter();

Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 100000; i++) {
        counter.increment();
    }
});

Thread thread2 = new Thread(() -> {
```

```
    for (int i = 0; i < 100000; i++) {
        counter.decrement();
    }
});

thread1.start();
thread2.start();
thread1.join();
thread2.join();

System.out.println("Final Counter Value: " + counter.getValue
());
```

- **Expected Result**: `0` (since increments and decrements cancel out).
- **Actual Result**: Unpredictable value due to race conditions.

6. Synchronization in Java

The `synchronized` Keyword

- **Purpose**: To prevent race conditions by allowing only one thread to execute a method at a time.
  - Enforces **mutual exclusion,** where any 2 methods cannot be executed at the same time.
- **Usage**: Add the `synchronized` keyword to all methods that must be made **mutually exclusive.**
  - Usually, every method that reads or writes field values should be synchronized.
  - Java will ensure that two synchronized methods of a given instance will ever be executed at the same time by different threads.

```
public synchronized void methodName() {
    // method body
```

```
}
```

Applying Synchronization to Counter Class

```
public class Counter {
    private int value = 0;

    public synchronized void increment() {
        value = value + 1;
    }

    public synchronized void decrement() {
        value = value - 1;
    }

    public synchronized int getValue() {
        return value;
    }
}
```

- **Effect**: Ensures mutual exclusion; only one thread can modify `value` at a time.

Locks and Mutual Exclusion

- **Lock Mechanism**: Each object has an intrinsic lock (monitor) associated with it.
    - When `synchronized` keyword is used, the JVM internally creates a **lock** for every instance of the class that is synchronized.
    - **Lock:** A tool for controlling access to a shared resource by multiple threads.
        - Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first.
- **When a `synchronized` method is called**:

- The calling thread acquires the object's lock.
- Other threads trying to call synchronized methods on the same object are blocked until the lock is released.

- **Releasing the Lock**:
  - Occurs when the synchronized method completes execution or throws an exception.

Manually using Java's `Lock` Instance

- Every instance that intends to use it needs to have its own lock.
  - Encapsulates the `Lock`, initializes it in the constructor with `lock = new ReentrantLock();`
  - In each method meant to be `synchronized` (i.e. mutually exclusive):
    - Acquire the lock, waiting if necessary until it is available.
    - Logic will only occur after the lock is acquired.
    - `unlock()` - Releases the lock - after the logic finishes.
    - `synchronized` keyword no longer needed.
  - **Example:**

```
public class Counter {
    private int value;
    private Lock lock;

    public Counter() {
        value = 0;
        lock = new ReentrantLock();
    }

    public void addOne() {
        lock.lock();
        value = getValue() + 1;
        lock.unlock();
```

```
    }
    public void subtractOne() {
        lock.lock();
        value = getValue() - 1;
        lock.unlock();
    }
    public int getValue() {
        lock.lock();
        int v = value;
        lock.unlock();
        return v;
    }
}

// => Is just how synchronized is internally
// implemented!
```

- **Best Practice:** Always ensure the lock is released, even with exceptions!
    - If used with a `try-catch` block, add a `finally` block to release it.
    - E.g.

    ```
    public int getValue() {
        lock.lock();
        try {
            return value;
        } finally {
            lock.unlock();
        }
    }
    ```

7. Deadlock Between Threads

Understanding Deadlocks

- **Deadlock**: A situation where two or more threads are blocked forever, each waiting for the other to release a lock.
- **Example Scenario**:
  - **Thread A** holds **Lock 1** and waits for **Lock 2**.
  - **Thread B** holds **Lock 2** and waits for **Lock 1**.
- **Result**: Neither thread can proceed.

Preventing Deadlocks

- **Lock Ordering**: Always acquire locks in a consistent order.
- **Timeouts**: Use timeouts when attempting to acquire locks.
- **Deadlock Detection**: Implement mechanisms to detect and resolve deadlocks.

8. Inter-Thread Communication

The `wait()` and `notify()` Methods

- **Purpose**: To coordinate execution between threads.
  - These are methods defined by `Object`.
- **Requirements**:
  - **Must be called within a synchronized context.**
    - i.e. calls to `wait()` or `notify()` must be within a `synchronized` method statement.
    - This is to ensure that the thread has acquired the lock on the object.
  - **The calling thread must own the object's lock.**

`wait()`

- **Behavior**: Causes the current thread to wait until another thread invokes `notify()` or `notifyAll()` on the same object.
  - The thread releases the lock on the object and goes into the **waiting state**.
- **Usage**:

```
public synchronized methodA(lockObject) {
    lockObject.wait();
}
```

- **Throws:**
    - `InterruptedException` if another thread interrupts the waiting thread.

`notify()`

- **Behavior**: Wakes up a single thread that is waiting on the object's monitor.
    - Releases one waiting thread (as soon as the lock is available).
    - The awakened thread will proceed when it regains the lock on the object.
- **Usage**:

```
public synchronized methodB(lockObject) {
    lockObject.notify();
}
```

`notifyAll()`

- **Behavior**: Wakes up all threads that are waiting on the object's monitor.

Example: Producer-Consumer Problem

```
class SharedResource {
    private int data = 0;
    private boolean available = false;

    public synchronized void produce(int value) {
        while (available) { // Wait if data is already availa
ble
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // Restor
```

```
e interrupted status
            }
        }
        data = value;
        available = true;
        System.out.println("Produced: " + value);
        notify(); // Notify a waiting consumer
    }

    public synchronized int consume() {
        while (!available) { // Wait if no data is available
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // Restor
e interrupted status
            }
        }
        available = false;
        System.out.println("Consumed: " + data);
        notify(); // Notify a waiting producer
        return data;
    }
}

class Producer extends Thread {
    private SharedResource resource;

    public Producer(SharedResource resource) {
        this.resource = resource;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            resource.produce(i);
        }
```

```java
    }
}

class Consumer extends Thread {
    private SharedResource resource;

    public Consumer(SharedResource resource) {
        this.resource = resource;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            resource.consume();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Producer producer = new Producer(resource);
        Consumer consumer = new Consumer(resource);

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

- **Explanation**:
    - **Producer** waits if the queue is full.
    - **Consumer** waits if the queue is empty.
    - `notifyAll()` wakes up waiting threads when the state changes.

9. Best Practices in Concurrent Programming

Lock Management

- **Always Release Locks**: Use `try...finally` blocks to ensure locks are released.

  ```
  lock.lock();
  try {
      // Critical section
  } finally {
      lock.unlock();
  }
  ```

- **Minimize Lock Scope**: Only lock the critical section, not the entire method.

Avoiding Common Pitfalls

- **Avoid Nested Locks**: Can lead to deadlocks.
- **Immutable Objects**: Prefer immutable objects when possible to avoid synchronization.
- **Use High-Level Concurrency Utilities**: Java provides classes like `ConcurrentHashMap`, `Semaphore`, and `CountDownLatch`.

Example: Using `ReentrantLock`

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class SafeCounter {
    private int value = 0;
    private Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            value = value + 1;
```

```
        } finally {
            lock.unlock();
        }
    }

    public int getValue() {
        lock.lock();
        try {
            return value;
        } finally {
            lock.unlock();
        }
    }
}
```

Additional Examples

Example 1: Simple Multithreading

Task: Print numbers from two threads.

```
public class MultiThreadExample {
    public static void main(String[] args) {
        Runnable printNumbers = () -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread " + Thread.current
Thread().getId() + ": " + i);
            }
        };

        Thread thread1 = new Thread(printNumbers);
        Thread thread2 = new Thread(printNumbers);

        thread1.start();
```

```
        thread2.start();
    }
}
```

- **Explanation**: Two threads execute the same task of printing numbers from 1 to 5.
- **Expected Output**:

  ```
  Thread 12: 1
  Thread 13: 1
  Thread 12: 2
  Thread 13: 2
  ...
  ```

Example 2: Demonstrating Race Condition

Task: Increment a shared variable without synchronization.

```
public class RaceConditionExample {
    public static void main(String[] args) throws Interrupted
Exception {
        class SharedCounter {
            public int count = 0;
        }

        SharedCounter counter = new SharedCounter();

        Runnable increment = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.count++;
            }
        };

        Thread thread1 = new Thread(increment);
```

```
        Thread thread2 = new Thread(increment);

        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();

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

- **Explanation**: The final count should be 2000 but may be less due to race conditions.

Example 3: Resolving Race Condition with Synchronization

Task: Increment a shared variable with synchronization.

```
public class SynchronizedCounterExample {
    public static void main(String[] args) throws Interrupted
Exception {
        class SharedCounter {
            private int count = 0;

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

            public int getCount() {
                return count;
            }
        }

        SharedCounter counter = new SharedCounter();
```

```
        Runnable increment = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread thread1 = new Thread(increment);
        Thread thread2 = new Thread(increment);

        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();

        System.out.println("Final Count: " + counter.getCount
());
    }
}
```

- **Explanation**: Synchronization ensures the final count is consistently 2000.