Concurrency in Java

1. Threads

- **Threads** are lightweight processes within a Java application that can run concurrently. They allow multiple tasks to execute simultaneously.
- Java provides built-in support for multithreading using the `Thread` class.
- To create a new thread, you can either extend the `Thread` class or implement the `Runnable` interface.

2. Methods in the `Thread` Class

The `Thread` class provides several important methods for managing threads:

1. **`start()`**: Initiates the execution of a thread. The `run()` method is called when the thread starts.

```
Thread thread = new Thread(new MyRunnable());
thread.start();
```

2. **`run()`**:Contains the code that constitutes the thread's task. Override this method when implementing custom thread behavior.

```
class MyThread extends Thread {
    public void run() {
        // Thread task implementation
    }
}
```

3. **`sleep(long millis)`**: Pauses the current thread for the specified duration (in milliseconds).

```
try {
        Thread.sleep(1000); // Sleep for 1 second
        } catch (InterruptedException e) {
        // Handle interrupted exception
        }
```

4. **`j`**oin():for the thread to terminate its execution before continuing with the current thread.

```
Thread thread = new Thread(new MyRunnable());
thread.start();
        try {
        thread.join(); // Wait for thread to finish
        } catch (InterruptedException e) {
        // Handle interrupted exception
        }
```

5. **`interrupt()`**: Interrupts the thread (useful for stopping long-running tasks).
6. **`isAlive()`**: Checks if the thread is still active.

3. Issues of Concurrency

1. **Race Conditions**: When multiple threads access shared resources concurrently, race conditions can occur. For example, if two threads increment a counter simultaneously, the result may be unexpected.
2. **Memory Consistency Errors**: In a multithreaded environment, changes made by one thread may not be visible to other threads due to caching and memory synchronization issues.
3. **Deadlocks**: Threads can get stuck waiting for each other indefinitely, preventing progress.
4. **Starvation**: A thread may be denied access to resources it needs due to other threads hogging them.

4. Synchronization

- **Synchronization** ensures that only one thread can access a critical section of code at a time.
- Java provides two ways to achieve synchronization:
    - **Method Synchronization**: Use the `synchronized` keyword to make a method thread-safe. Example:

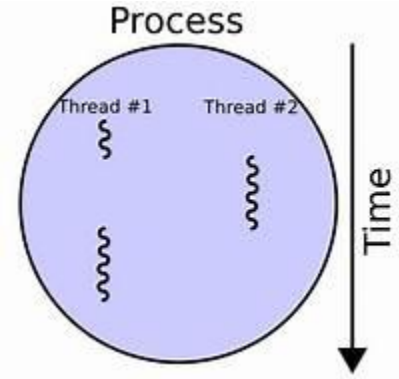
```
class Counter {
private int count = 0;

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

- **Block Synchronization**: Use synchronized blocks to protect specific code sections. Example:

```
class SharedResource {
    private final Object lock = new Object();

    public void doSomething() {
        synchronized (lock) {
            // Critical section
            // ...
        }
    }
}
```



Multithreading is a model of program execution that allows for multiple threads to be created within a process, executing independently but concurrently sharing process resources. Each thread is a self-contained sequence of instructions that can execute in parallel with other threads that are part of the same root process. In other words:

- **Threads** are lightweight processes within a program.
- They run concurrently, sharing the same resources (such as CPU cores, caches, and memory).
- Multithreading aims to increase utilization of a single core by using thread-level parallelism, allowing multiple tasks to execute simultaneously1.

In Java, multithreading is a powerful feature that allows you to create and manage threads for maximum CPU utilization. Threads can be created using different mechanisms, such as extending the Thread class or implementing the Runnable interface.

Example 1: Extending the `Thread` Class

In this example, we'll create a simple program that spawns two threads to print numbers from 1 to 10 concurrently.

```
class NumberPrinter extends Thread {
    private final int start;
    private final int end;

    public NumberPrinter(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public void run() {
        for (int i = start; i <= end; i++) {
            System.out.println(Thread.currentThread().getName() + ": " + i);
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        NumberPrinter thread1 = new NumberPrinter(1, 5);
        NumberPrinter thread2 = new NumberPrinter(6, 10);

        thread1.start();
        thread2.start();
    }
}
```

Example 2: Implementing the `Runnable` Interface

In this example, we'll achieve the same result by implementing the `Runnable` interface.

```
class NumberPrinterRunnable implements Runnable {
    private final int start;
    private final int end;

    public NumberPrinterRunnable(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public void run() {
        for (int i = start; i <= end; i++) {
            System.out.println(Thread.currentThread().getName() + ": " + i);
        }
    }
```

```
}

public class RunnableExample {
    public static void main(String[] args) {
        NumberPrinterRunnable task1 = new NumberPrinterRunnable(1, 5);
        NumberPrinterRunnable task2 = new NumberPrinterRunnable(6, 10);

        Thread thread1 = new Thread(task1);
        Thread thread2 = new Thread(task2);

        thread1.start();
        thread2.start();
    }
}
```

Remember that the output order may vary due to thread scheduling. You'll see interleaved numbers printed by both threads.

In Java, **daemon threads** are low-priority threads that run in the background to perform specific tasks. Let's explore some key points about daemon threads:

1. **Purpose of Daemon Threads**:
    - Daemon threads serve user threads by handling background tasks without interfering with the main execution.
    - Examples of daemon threads include garbage collection (gc) and finalizer threads.
2. **Characteristics of Daemon Threads**:
    - **Preventing JVM Exit**: Daemon threads cannot prevent the Java Virtual Machine (JVM) from exiting when all user threads finish their execution. If all user threads complete their tasks, the JVM terminates itself, regardless of whether any daemon threads are running.
    - **Automatic Termination**: If the JVM detects a running daemon thread, it terminates the thread and subsequently shuts it down. The JVM does not check if the daemon thread is actively running; it terminates it regardless.
    - **Low Priority**: Daemon threads have the lowest priority among all threads in Java.
3. **Default Nature of Daemon Threads**:
    - By default, the main thread is always a non-daemon thread.
    - For all other threads, their daemon nature is inherited from their parent thread. If the parent thread is a daemon, the child thread is also a daemon; if the parent thread is a non-daemon, the child thread is also a non-daemon.
4. **Methods for Daemon Threads**:
    - `setDaemon(boolean status)`: Marks the current thread as a daemon thread or user thread. Setting a user thread as a daemon can be done using

`tU.setDaemon(true)`, while setting a daemon thread as a user thread can be done using `tD.setDaemon(false)`.

- `isDaemon()`: Checks if the current thread is a daemon. It returns true if the thread is a daemon, otherwise false1.

Remember that daemon threads are essential for background tasks and are automatically terminated when user threads complete their execution.