Introduction to Multithreading

Multithreading is a Java feature that allows the execution of multiple threads simultaneously. Threads are lightweight subprocesses that share the same memory space, making them more efficient than multitasking processes.

- **Use Case**: Useful for tasks like file I/O, computational tasks, animations, or handling multiple user requests in web applications.
- **Key Benefits**:
  - Better CPU utilization.
  - Faster execution by running tasks in parallel.
  - Simplifies modeling of real-world systems

Key Concepts

1. **Thread**: A unit of a process that executes code.
2. **Multithreading**: Running multiple threads concurrently.
3. **Concurrency vs Parallelism**:
   - **Concurrency**: Multiple threads making progress simultaneously.
   - **Parallelism**: Threads actually running simultaneously on multi-core CPUs.

Thread Lifecycle

1. **New**: Thread object created but not started.
2. **Runnable**: Thread is ready to run but waiting for CPU scheduling.
3. **Running**: Thread is executing.
4. **Blocked/Waiting**: Thread is paused or waiting for resources.
5. **Terminated**: Thread has completed execution.

How to Create Threads

1. Extending the `Thread` class

```
class MyThread extends Thread
{
    public void run(){
        System.out.println("Thread is running");
    }
}
public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start(); // Starts the thread
    }
}
```

2. Implementing the `Runnable` interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

3. Using Lambda Expression (since Java 8)

```
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(() -> System.out.println("Thread is running"));
        t.start();
    }
}
```

Thread Methods

Method	Description
`start()`	Starts a thread.
`run()`	Contains the thread's execution logic.
`sleep(ms)`	Makes a thread sleep for a specified time.
`join()`	Waits for a thread to die.
`getName()`	Gets the thread's name.
`setName(String)`	Sets the thread's name.
`isAlive()`	Checks if the thread is alive.
`setPriority(int)`	Sets thread priority (`1` to `10`).
`getPriority()`	Gets the thread priority.
`yield()`	Temporarily pauses and allows other threads to execute.

Thread Synchronization

To prevent thread interference and ensure thread safety, use **synchronized blocks** or methods.

Synchronized Method

```
class Counter {
    synchronized void increment() {
        // Critical section
    }
```

```
}
```

Synchronized Block

```
class Counter {

    void increment() {
        synchronized (this) {
            // Critical section
        }
    }
}
```

Inter-thread Communication

1. **`wait()`**: Causes a thread to wait until another thread invokes `notify()` or `notifyAll()`.
2. **`notify()`**: Wakes up a single waiting thread.
3. **`notifyAll()`**: Wakes up all waiting threads.

Example:

```
class SharedResource {
    synchronized void produce() throws InterruptedException {
        wait();
        System.out.println("Producing...");
    }

    synchronized void consume() {
        System.out.println("Consuming...");
        notify();
    }
}
```

Deadlock

Occurs when two or more threads block each other, waiting for a resource. To avoid:

1. Use consistent resource ordering.
2. Avoid nested locks.

Thread Pooling

Instead of creating new threads for each task, use a **thread pool** to reuse existing threads.

Example using `ExecutorService`:

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

```
public class Main {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 5; i++) {
            executor.execute(() -> System.out.println("Task executed by " +
Thread.currentThread().getName()));
        }
        executor.shutdown();
    }
}
```

Concurrency Utilities

Java provides the `java.util.concurrent` package for high-level concurrency:

1. **Locks** (`ReentrantLock`, `ReadWriteLock`).
2. **Atomic Variables** (`AtomicInteger`, `AtomicBoolean`).
3. **CountdownLatch**: Allows one or more threads to wait until a set of operations are completed.
4. **CyclicBarrier**: Allows threads to wait at a barrier point.

Example using `ReentrantLock`:

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

class Shared {
    private final ReentrantLock lock = new ReentrantLock();

    void access() {
        lock.lock();
        try {
            System.out.println("Accessed by " +
Thread.currentThread().getName());
        } finally {
            lock.unlock();
        }
    }
}
```