Object Oriented Programming in Java

Introduction to Multithreading and Concurrency

Suhel Hammoud

What is Concurrency?

- Concurrency is the ability to run several programs or parts of a program in parallel.
- In Java, concurrency is achieved through:
  - Threads
  - Runnable interface
  - Executors
  - Concurrency utilities in `java.util.concurrent`

Thread Basics

Creating a Thread

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}

MyThread t = new MyThread();
t.start();
```

Or using `Runnable`:

```
Runnable r = () -> System.out.println("Runnable running");
new Thread(r).start();
```

▒▒ Thread Lifecycle

1. New
2. Runnable
3. Running
4. Blocked/Waiting
5. Terminated

Use `start()`, `sleep()`, `join()`, `wait()`, and `notify()` to manage thread behavior.

▓▓ Thread Synchronization

▒▒ Why Synchronize?

To avoid race conditions when multiple threads access shared data.

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

Use `synchronized` blocks or methods.

▒▒ Volatile Keyword

- Ensures visibility of changes to variables across threads.

```
volatile boolean flag = true;
```

- Does **not** ensure atomicity.

▒▒ Locks and ReentrantLock

- Alternative to synchronized

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
}
```

▒▒ Executor Framework

- Better thread management

```
ExecutorService executor = Executors.newFixedThreadPool(3);
executor.submit(() -> {
    System.out.println("Task executed");
});
executor.shutdown();
```

▒▒ Callable and Future

- Callable returns a value

```
Callable<Integer> task = () -> 123;
Future<Integer> future = executor.submit(task);
Integer result = future.get(); // blocks until done
```

▒▒ Thread-safe Collections

- Use `ConcurrentHashMap`, `CopyOnWriteArrayList`, etc.

```
Map<String, Integer> map = new ConcurrentHashMap<>();
```

These are designed for high-concurrency scenarios.

▒▒ Atomic Variables

- Operations like increment are atomic

```
AtomicInteger count = new AtomicInteger(0);
count.incrementAndGet();
```

Part of `java.util.concurrent.atomic`

CountDownLatch & CyclicBarrier

CountDownLatch

```
CountDownLatch latch = new CountDownLatch(3);
latch.countDown();
latch.await();
```

CyclicBarrier

```
CyclicBarrier barrier = new CyclicBarrier(3, () -> System.out.println("All
threads reached barrier"));
```

▒▒ Fork/Join Framework

- Recursive task division

```
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new MyRecursiveTask());
```

Use `RecursiveTask<T>` or `RecursiveAction`.

▒▒ Best Practices

- Avoid shared mutable state
- Prefer higher-level abstractions (Executors, Futures)
- Use thread-safe collections and atomics
- Monitor performance and deadlocks

▒▒ Resources

- **Official Oracle Java Docs** (https://docs.oracle.com/javase/tutorial/essential/concurrency/)