Advanced Multithreading

ReentrantLock

- ReentrantLock extends functionality of synchronization keyword.
- ReentrantLock is added in Java 1.5.
- ReentrantLock provides more control on lock acquisition.

ReentrantLock code

```
public synchronized double deposit(double amount) {
        balance = balance + amount;
        return balance;
}
ReentrantLock lock = new ReentrantLock();
public double deposit(double amount) {
        lock.lock();
        try {
                balance = balance + amount;
        }
        finally {
                lock.unlock();
        }
}
```

Why ReentrantLock?

- synchronized does not have fairness supported. ReentrantLock has support for fairness. It means we can specify that longest waiting thread will get the preference.
- Reentrant lock has tryLock() & tryLock(time) methods which checks weather the lock is available or not.
- ReentrantLock can provide us the list of the threads waiting for the lock.

Limitations of ReentrantLock

- Need to wrap the method body inside try/finally blocks which makes code unreadable & hides business logic.
- It is developer's responsibility to acquiring & releasing the lock. If we forget to release the lock then it leads to major bug difficult to analyse.

Automic operations

```
public int getCount() {
        return count++;
}
```

```
AtomicInteger count = new AutomicInteger();
public int getCount() {
        return count.incrementAndGet();
}
```

Lambda expressions in Multithreading

```
Runnable runnable = () -> { System.out.println("Inside run"); };
Thread t = new Thread(runnable);
t.start();
```

Java Executor framework

- Executor framework was introduced in Java 1.5 concurrency API.
- Executor framework provides us high level replacement for working with threads directly.
- Executor framework is capable of running asynchronous tasks and typically manage a pool of threads.
- In Executor framework, programmer does not need to create the thread manually.

Applying Executors

```
ExecutorService executor = Executors.newFixedThreadPool(3);
Runnable runnable = new Runnable() {
        public void run() {
                System.out.println("Thread running");
        }
};
executor.execute(runnable);
```

Executor methods

- newSingleThreadExecutor()

  Creates an Executor that uses a single worker thread.

- newFixedThreadPool(int nThreads)

  Creates a thread pool that reuses a fixed number of threads.

- newCachedThreadPool()

  Creates a thread pool that creates new threads as needed.

Using Callable

```
ExecutorService executor = Executors.newFixedThreadPool(3);
Callable<Integer> callable =  new Callable<Integer>(){
        @Override
        public Integer call() throws Exception {
                System.out.println("Callable Thread started");
                return 1;
        }
};
Future<Integer> future = executor.submit(callable);
int result = future.get();
```

Callables & Futures

- In addition to Runnable, executor framework supports another kind of task named 'Callable.
- Runnable returns void but Callable returns a value.
- Callables are submitted to executors with submit() method instead of execute().
- Callable return value can be accessed using a special object called 'Future'.