__/__/____

Multithreading

Multitasking :-

Multitasking allows several activities to occur concurrently on the computer.

(i) Process-based multitasking.
(ii) Thread-based multitasking.

Process Based Multitasking :-

Allows processes (i.e. programs) to run concurrently on the computer.
Eg- Running the MS Paint while also working with the processor.

Thread Based Multitasking :-

Allows parts of the same program to run concurrently on the computer.
Eg- MS word that is printing and formatting text at the same time.

* Threads VS Process :-

→ Two threads share the same address space.

→ context switching between threads is usually less expensive than between process.

__/__/____

→ The cost of communication between threads is relatively low.

Why Multithreading ?

→ In a single-threaded environment, only one task at a time can be performed.

→ CPU cycles are wasted, for example, when waiting for user input.

→ Multitasking allows idle CPU time to be put to good use.

Threads :-

A thread is an independent sequential path of execution within a program.

→ Many threads can run concurrently within a program.

→ At runtime, therefore, share both data and code (i.e., they are lightweight compared to processes).

__/__/____

→ 3 Important Concepts related to Multithreading in Java :-

1. Creating threads and providing the code that gets executed by a thread.
2. Accessing common data and code through synchronization.
3. Transitioning between thread states.

Main Thread :-

When a standalone application is run, a user thread is automatically created to execute the main() method of the application. This thread is called the main thread.

If no other user threads are spawned, the program terminates when the main() method finishes executing.

All other threads, called threads, are spawned from the main thread.

The main() method can then finish, but the program will keep running until all user threads have completed.

__/__/____

The runtime environment distinguishes btw user threads and demon threads.

Calling the setDaemon(boolean) method in the Thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started.

As long as a user thread is alive, the JVM does not terminate.

A daemon thread is at the mercy of the runtime system: it is stopped if there are no more user threads running, thus terminating the program.

__/__/___

Thread creation :

A thread in Java is represented by an object of the Thread class.

Creating threads is achieved in one of two way.

1. Implementing the Java.lang.Runnable interface.
2. Extending the Java.lang.Thread class.

* we are creating thread using implementing the Java.lang.Runnable interface because we can implement multiple interface but in Java there is no concept of multiple inheritance so if we have to inherit the class we can't do because we already extends the Thread class.

__/__/____

Synchronization :-

Thread share the same memory space, i.e. they can share resources (objects).

→ However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource.

Synchronized Methods :-

While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait.

This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object.

Such a method can invoke other synchronized methods of the object without being blocked

__/__/____

Rules of Synchronization :-

A thread must acquire the object lock associated with a shared resource before it can enter the shared resource.

The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with it.

If a thread cannot immediately acquire the object lock, it is blocked, i.e., it must wait for the lock to become available.

When a thread exists a shared resource, the runtime system ensures that the object lock is also relinquished. If another thread is waiting for this object lock, it can try to acquire the lock in order to gain access to the shared resource.

It should be made clear that programs should not make any asumptions about the order in which threads are granted ownership of a lock.

__/__/____

Static Synchronized Methods :-

A thread acquiring the lock of a class to execute a static synchronized method has no effect on any thread acquiring the lock on any object of the class to execute a synchronized instance method.

→ In other words, synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.

→ A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass.

Race Condition :-

It occurs when two or more threads simultaneously update the same value (StackTopIndex) and, as a consequence, leave the value in an undefined or inconsistent state.

__/__/____

Synchronized Blocks :-

- Whereas execution of synchronized methods of an object is synchronized on the lock of the object, the synchronized block allows execution of arbitrary code to be synchronized statement is as follows.

→ The general form of the synchronized statement is as follows :-

synchronized (object ref expression) {<code block>}

→ The object ref expression must evaluate to a non-null reference value, otherwise a NULL-PointerException is thrown.

Summary :-

A thread can hold a lock on an object :

→ By executing a synchronized instance method of the object. (this).

→ By executing a synchronized static method of a class or a block inside a static method (in which case, the object is the class object representing the class in the JVM)

__/__/____

* Thread safety :-

It's the term used to describe the design of classes that ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads. Eg. StringBuffer

The Volatile Keyword :-

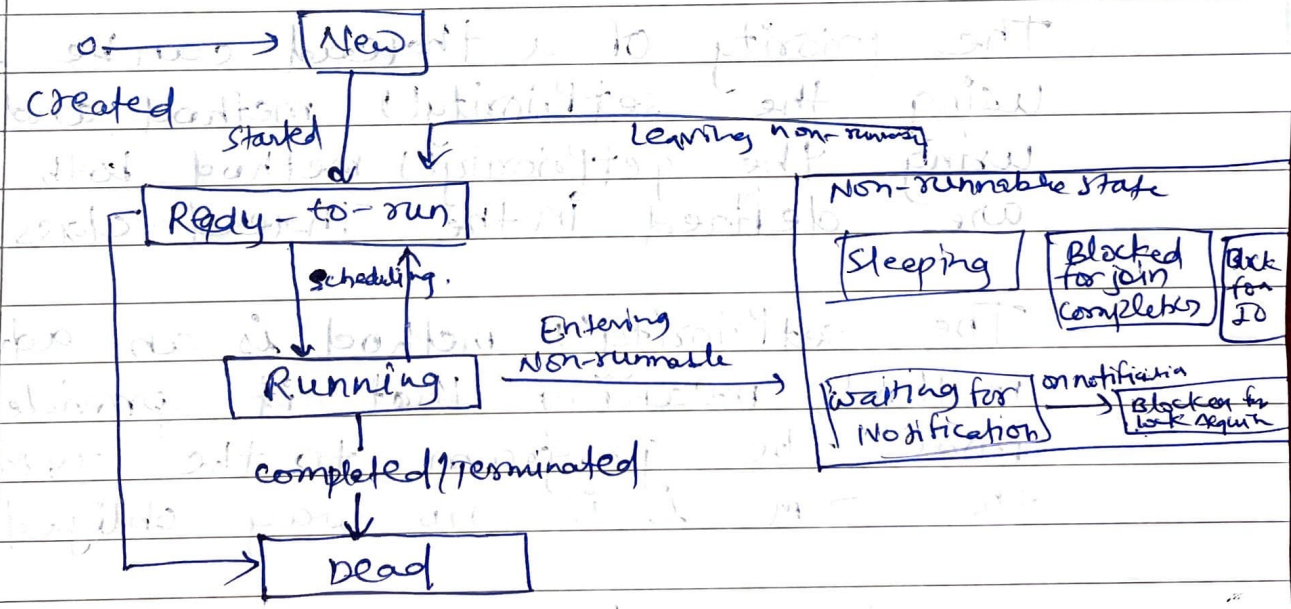
The volatile keyword in Java is used to indicate that a variable's value may be modified by multiple threads simultaneously. It ensures that the variable is always read from and written to the main memory, rather than from thread-specific caches, ensuring visibility across threads.

__/__/____

Producer Consumer Pattern :-

The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Thread State :-



__/__/____

Thread Priorities :-

Priorities are integer values from 1 (lowest priority given by the constant Thread.MIN_PRIORITY) to 10 (highest priority given by the constant Thread.MAX_PRIORITY). The default priority is 5 (Thread.NORM_PRIORITY).

A thread inherits the priority of its parent thread.

The priority of a thread can be set using the setPriority() method and read using the getPriority() method, both of which are defined in the Thread class.

The setPriority() method is an advisory method, meaning that it provides a hint from the program to the JVM, which the JVM is in no way obliged to honor.

Thread Scheduler :-

Schedulers in JVM implementations usually employ one of the two following strategies

1. Preemptive scheduling.
2. Time-sliced or Round-Robin scheduling.

__/__/__

Deadlocks :-

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds

→ Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever in the Blocked for-lock-acquistion state.

Java :-

```
String lock1 = "ABC";
String lock2 = "DEF";

Thread thread1 = new Thread () -> {
        Synchronized (lock1){
         try {
              Thread.sleep(1);
         }catch (InterruptedException e) {
              e.printStackTrace();
         }
         Synchronized (lock2) {
              System.out.println ("lock acqui");
         }
        }
    }, ("Thread1");
```

__/__/____

```
Thread thread2 = new Thread(() -> {
    synchronized (lock2) {
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (lock1) {
            System.out.println("lock acquired");
        }
    }
}, "thread2");
thread1.start(); thread2.start();
```