Multithreading & Concurrency in Java

Prepared by: Vishal Arya

Multitasking - multitasking allows several activities to run concurrently on the computer.

Eg. A user working on the notepad, a web browser and a music player. The user can type in the notepad, browse the internet on the web browser while listen to the music in the music player without any delay or interruption.

- process-based multitasking
- Thread-based multitasking

Process-based multitasking - it allows multiple programs to run parallelly.
Eg. in a desktop environment each application runs as a separate process, allowing users to run multiple applications simultaneously and switching between them seamlessly.

Thread based multitasking - Thread based multitasking allows for concurrent execution of multiple task within a single program or process.
Eg. In the video player application, one thread might be responsible for playing the video while another thread handles user input and controls the playback.

Thread

- A thread is an independent sequential path of execution within a program.
- Many threads can run currently within a program.
- At runtime, threads is a program exist in a common memory space and can, therefore share both data and code i.e they are lightweight compared to process.
- They also share the process running the program.

Process	Thread
A process is an instance of a program that is being executed or processed	Thread is a segment of process or a lightweight process that is managed by the scheduler.
Processes are independent of each other, and hence don't share memory or other resources	Threads are interdependent and share memory.
Context switching between two processes take much time	Context switching between threads usually less expensive because they are very lightweight.
The process is less efficient in terms of communication	Cost of communication between thread is relatively Low.

Why multithreading?

- In a single threaded environment, only one task at a time can be performed.
- CPU cycle or wasted for example when waiting for user input.
- Multitasking allows ideal CPU time to be put to good use.

Main Thread

- When a standalone application is run, a user thread is automatically created to execute the main() method of application it is called main thread.
- If no other user threads are spawned, the program terminates when the main() method finishes executing.
- All other thread are called child thread, are spawned from the main thread.
- The main() method can then finish, but the program will keep running until all the user threads have completed.
- The runtime environment distinguishes between user threads and daemon 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 has 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. There are two ways to create thread:

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

Better to create a thread by implementing the runnable interface because Java support implementing multiple interface.

On the other hand, if we are creating the thread by extending the Thread class, then at some point you want to extend another class then it's not possible because Java doesn't support multiple inheritance.

Synchronization

- Thread share the same memory space i.e they can share resources (objects).
- However there are critical situation where it is desirable that only one thread at a time has access to a shared resources.

Eg. Without synchronization, multiple users can book the same ticket simultaneously, resulting in inconsistent data and incorrect booking status. Consider the following scenario.

- User A and User B both want to book ticket1.
- User A sends request to book ticket1.
- User B sends request to book ticket1.
- The BookingSystem creates two threads to process the request.
- Thread 1 (for User A) reserves ticket1.
- Thread 2 (for User B) reserves ticket1.
- Both threads return successfully, indicating that the ticket has been booked.

In this scenario, both users were able to book the same ticket simultaneously, leading to inconsistent data and incorrect booking status. To prevent this, we can use synchronization to ensure that only one thread can access the ticket book method at a time.

Checkout the repo link for example:
github.com/vishalarya01/multithreading-in-java

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 locked.
- The non-synchronized methods of the object can always be called at any time by any thread.

Rules of synchronization:

- A thread must acquire the object look associated with a shared resource before it can enter the shared resource.
- The runtime system ensures that no other thread can enter a shared resources if another thread already holds the object lock associated with it.
- If a thread cannot immediately acquire the object lock, it is blocked, that is it must wait for the lock to become available.

- When a thread exits a shared resource, the runtime system ensures that the object lock is also relinquished. If another thread is awaiting 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 assumptions 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 another words, synchronization of static method 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 and, as a consequence, leave the value in an undefined or inconsistent state.

Eg. Suppose there is a account with balance 1000rs. Two concurrent transactions are being processed:

1. T1: withdrawal of 500rs from the account.
2. T2: deposit of 200rs into the account

If both transactions execute simultaneously, following sequence of events may occur:

T1: reads account balance as 1000rs.
T2: reads account balance as 1000rs.
T2: add 200rs to the balance and writes the new balance as 1200rs.
T1: subtracts 500rs from the balance and writes the new balance as 500rs.

The final Balance is now 500rs but correct balance should be 700rs.

This is the classic example of race condition where the final outcome depends on the order in which the transactions are executed.
To avoid this situation, the application can use synchronization mechanisms such as lock to ensure that only one transaction can access the account balance at a time. This will ensure that the transactions are executed in the core order and final balance of the account will have correct value of 700rs.

Synchronized Blocks:

- Whereas execution of synchronized methods of an object is synchronized on the lock of the object, the synchronized blocks allows execution of arbitrary code to be synchronized on the lock of an arbitrary object.
- synchronized (object ref expression) {...code..}
- The object ref expression must evaluate to be a non-null reference value, otherwise a NullPointerException is thrown.

Summary:

A thread can hold a lock on an object:

- By executing a synchronized instance method of the object. (this)
- By executing the body of a synchronized block that synchronizes on 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

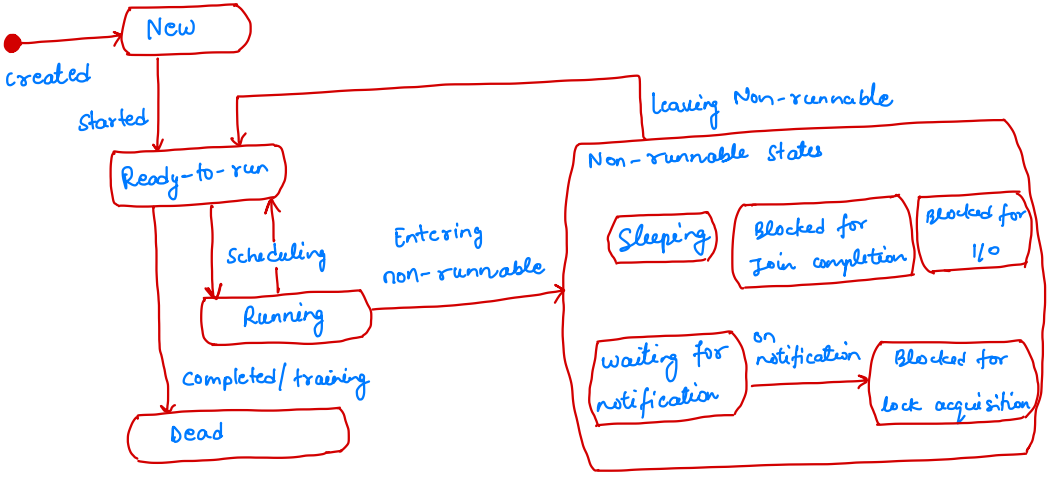
It is the term used to describe the design of classes that ensure that the state of their objects is always consistent even when the object are used concurrently by multiple threads.

Eg. StringBuffer

Thread priority

- Threads are assigned priorities that the thread scheduler can use to determine how the threads will be scheduled.
- The thread scheduler can use thread priority to determine which threads gets to run.
- Priority level is 1 - 10, 1 is the lowest and 10 is highest priority.
- Default priority is 5.
- Thread.NORM_PRIORITY, Thread.MIN_PRIORITY, Thread.MAX_PRIORITY.
- A Thread inherits the priority of its parent thread.
- Priority can be set using setPriority() method and read using getPriority() method.
- The setPriority() method is advisory method, meaning that it provides a hint from the program to JVM, which the JVM is in no way obliged to honor.

Thread states



Constant in the Thread.State enum type	State	Description of the thread
NEW	New	Created but not yet started
RUNNABLE	Runnable	Executing in the JVM
BLOCKED	Blocked for aquisition	Blocked while waiting for a Lock
WAITING	Waiting for notify, Blocked for join completion	Waiting indefinitely for another thread to perform a particular action
TIMED_WAITING	Sleeping, waiting for notify, Blocked for join completion	Waiting for another thread to perform an action for up to a specified time
TERMINATED	Dead	Completed execution