MultiThreading

Thread Lifecycle:

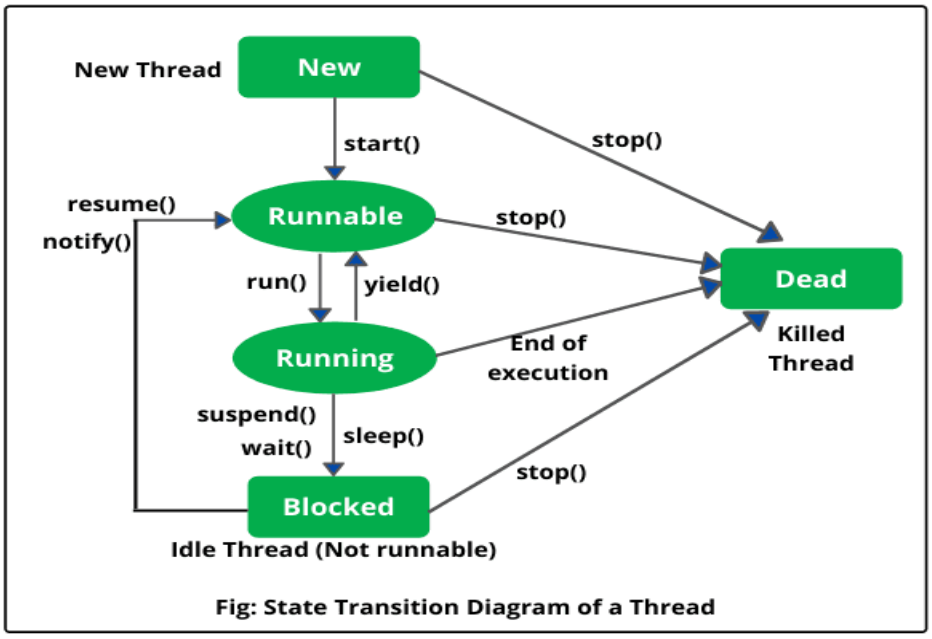

Fig: State Transition Diagram of a Thread

- New state: Occurs when a thread is created but not yet started.
- Runnable state: Occurs after the start() method is invoked, and the thread is alive and either running or waiting for resource allocation.
- Blocked state: Occurs when a thread is temporarily inactive and waiting to acquire a monitor lock to access a synchronized block or method.
- Waiting state: Occurs when a thread is waiting for another thread to perform a specific action.
- Terminated state: Occurs when a thread has finished execution.

Thread.sleep:

Thread.sleep() method in Java is used to pause the execution of the current thread for a specified period of time. When the current thread in sleep gets interrupted by another thread, it throws an InterruptedException. When the specified sleep time during the function call is negative, it throws an IllegalArguementException.

Join:

java.lang.Thread class provides the join() method which allows one thread to wait until another thread completes its execution. If t is a Thread object whose thread is currently executing, then t.join() will make sure that t is terminated before the next instruction is executed by the program.

In Java, the wait() method is used to pause the execution of a thread until another thread signals that it can resume. When a thread calls wait() on an object, it releases the lock on the object and waits until another thread calls notify() or notifyAll() on the same object.

Syntax

```
public final void wait(long timeout, int nanos) throws InterruptedException
synchronized(object)
{
   while(condition is false)
   {
      object.wait();
   }
       //do the task
}
```

Note:

1. IllegalArgumentException- This is the exception thrown when the value of the arguments is not in the acceptable range (negative or out of bounds).
2. IllegalMonitorStateException- The exception thrown if the thread being executed does not have permission to monitor the object's state.
3. InterruptedException- The exception is thrown when another thread interrupts the execution of the current thread. Once this exception is thrown, the interruption clears.

InterThread Communication:

Inter Thread Communication is a method that allows many synchronized threads to communicate or interact with one another.

In Java, there are two ways to implement inter-thread communication:
using wait() and notify() methods and using the higher-level constructs of the java.util.concurrent package.

Inter-Thread Communication (Cooperation) is a mechanism that allows threads to exchange information or coordinate their execution. It enables threads to work together to solve a common problem or to share resources.

Wait() Method

Let's say we are currently running Thread1 and we want to run Thread2. Since inter thread communication can be done in a synchronized block only one thread can run at a time. So to run Thread2 we must "pause" Thread1. The wait() function helps us achieve this exact thing.

The wait() method aids inter thread communication by releasing the lock on the current or calling thread and instructing it to sleep until another thread enters the monitor and calls notify() or notifyAll(), or until a certain period of time has passed.

The current thread must own this object's monitor, hence the wait() function must be used from the synchronized method only; otherwise, an error will be thrown.

public final void wait(long timeout, int nanos) throws InterruptedException

Notify() Method

Now let's say after we completed our work in Thread2 and we want to return to Thread1. To do so, we need to inform Thread1 that it can now use the object to run itself. We can use the notify() method to do so.

NotifyAll() Method

In a similar way to the notify() method, the notifyAll() method aids inter thread communication. The only difference is that it wakes up all of the object monitor waiting for threads, making them all runnable at the same time.

ThreadPool:

A thread pool is a collection of threads that are managed by a thread pool manager. When a task is submitted to the thread pool, it assigns the task to an available thread from the pool.

This way, we can limit the number of threads in our application and prevent it from creating too many threads, which can cause performance issues.

There are 4 types of Threads Pools in java

1) Cached Thread Pool

2) Fixed Thread Pool

3) Single Thread Executor

4) Scheduled Thread Pool

Cached Thread Pool

- If we create a lot of threads in the pool, we will face high memory utilization and degraded performance of the application
- If we create too few threads, we would not get the benefits of using a thread pool

Java has an interesting way of creating a Thread pool
using Executors.newChachedThreadPool(). This implementation creates an initial pool of zero threads, whenever there is a demand, more threads are created and added to the pool.

Threads which remain idle for more than 60 seconds are removed from the pool

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

public class CachedThreadPoolExample {

   public static void main(String[] args) {
      ExecutorService executor = Executors.newCachedThreadPool();

      for (int i = 0; i < 10; i++) {
         executor.execute(() -> {
            System.out.println("Task is running.");
         });
      }

      executor.shutdown();
   }}
```

Fixed Thread Pool

The Fixed Thread Pool has a fixed number of threads that are created when the thread pool is initialized. Once a thread is created, it remains in the pool until the thread pool is shut down. If a new task is submitted and all the threads are busy, the task is added to a queue and waits for an available thread.

```
ExecutorService executor = Executors.newFixedThreadPool(5);
```

Single Thread Executor

The Single Thread Executor has only one thread in the pool. It executes tasks sequentially, one at a time. If a task is submitted to the thread pool and the thread is busy, the task is added to a queue and waits for the thread to become available

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

Note : Formula to compute Thread Pool Size:

Numberofthreads=NumberofAvailableCores∗(1+Waittime/Servicetime)

Scheduled Thread Pool

The Scheduled Thread Pool is similar to the Fixed Thread Pool, but it is designed for executing tasks at a specific time or repeatedly at a fixed interval. You can use it to schedule tasks to run at a certain time or to repeat at a certain interval.

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);

    executor.schedule(() -> {
        System.out.println("Task is running after 5 seconds.");
    }, 5, TimeUnit.SECONDS);

    executor.scheduleAtFixedRate(() -> {
        System.out.println("Task is running repeatedly every 2 seconds.");
    }, 0, 2, TimeUnit.SECONDS);

    executor.shutdown();
```

ThreadGroup:

Thread group in java is used to group similar threads into one unit. A thread group can also contain other thread groups. Thread groups are constructed using java.lang.ThreadGroup class.

The main use of thread groups is that you can handle multiple threads simultaneously.

Example:

```
ThreadGroup parentGroup = new ThreadGroup("Parent Thread Group");

    //Adding threads to ThreadGroup while creating threads itself

    Thread t1 = new Thread(parentGroup, "Thread 1");

    Thread t2 = new Thread(parentGroup, "Thread 2");
```

Synchronization:

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

There are 2 types of Synchronizations in java

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package **java.util.concurrent.locks** contains several lock implementations.

Synchronized blocks:

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

Syntax

```
synchronized (object reference expression) {
  //code block
}
```

Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException.

If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.

Future:

A Future interface provides methods to check if the computation is complete, to wait for its completion and to retrieve the results of the computation.

The result is retrieved using Future's get() method when the computation has completed, and it blocks until it is completed.

Future and FutureTask both are available in java.util.concurrent package from Java 1.5.

Note:

1. FutureTask is a concrete implementation of the Future, Runnable, and RunnableFuture interfaces and therefore can be submitted to an ExecutorService instance for execution.
2. When calling ExecutorService.submit() on a Callable or Runnable instance, the ExecutorService returns a Future representing the task. and one can create it manually also.

Methods in future interface:

Method	Description
cancel()	It tries to cancel the execution of the task.
get()	The method waits if necessary, for the computation to complete, and then retrieves its result.
get()	Waits if necessary, for at most the given time for the computation to complete, and then retrieves its result, if available.
isCancelled()	It returns true if the task was cancelled before its completion.
isDone()	It returns true if the task is completed.

Executor Service:

The Java ExecutorService interface is present in the java.util.concurrent package. The ExecutorService helps in maintaining a pool of threads and assigns them tasks.

It also provides the facility to queue up tasks until there is a free thread available if the number of tasks is more than the threads available.

Example:

```
public class ExecutorServiceExample {

   public static void main(String[] args) {
      ExecutorService executorService = Executors.newFixedThreadPool(10);
      executorService.execute(new Runnable() {

         @Override
         public void run() {
            System.out.println("ExecutorService");

         }
      });
      executorService.shutdown();
   } }
```

Example for future:

```
Future future = ... // get Future by starting async task

// do something else, until ready to check result via Future

// get result from Future
try {
   Object result = future.get();
} catch (InterruptedException e) {
   e.printStackTrace();
} catch (ExecutionException e) {
   e.printStackTrace();
}
```

2)

```
import java.util.concurrent.*;
public class Main {
   public static void main(String[] args) throws ExecutionException, InterruptedException {
      ExecutorService executor = Executors.newSingleThreadExecutor();
     Future<Integer> future = executor.submit(new Callable<Integer>() {
         @Override
         public Integer call() throws Exception {
            int sum = 0;
            for (int i = 1; i <= 100; i++) {
```

```
                sum += i;
            }
            return sum;
        }
    });
    System.out.println("Waiting for the result...");
    int result = future.get();
    System.out.println("The sum of the first 100 numbers is: " + result);
    executor.shutdown();
  }
}
```