Threads

* A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources.
* Multitasking is when multiple processes share common processing resources such as a CPU.
* Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel.

**Advantage of Java Multithreading**

1) It doesn't block the user because threads are independent and you can perform multiple operations at same time.

2) You can perform many operations together so it saves time.

3) Threads are independent so it doesn't affect other threads if exception occur in a single thread.

**What is a thread?**

* A thread is a lightweight sub process, a smallest unit of processing.
* It is a separate path of execution.
* Threads are independent, if there occurs exception in one thread, it doesn't affect other threads.
* It shares a common memory area.
* Thread is executed inside the process.

**Life cycle of a Thread (Thread States):**

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

* **New** - The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
* **Runnable** - The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
* **Running** - The thread is in running state if the thread scheduler has selected it.
* **Non-Runnable (Blocked)** - This is the state when the thread is still alive, but is currently not eligible to run.
* **Terminated** - A thread is in terminated or dead state when its run() method exits.

**Thread Creation:**

There are two ways to create a thread:

* By extending Thread class
* By implementing Runnable interface.

**Thread class:**

* Thread class provide constructors and methods to create and perform operations on a thread.
* Thread class extends Object class and implements Runnable interface.

**Constructors of Thread class:**

* Thread()
* Thread(String name)
* Thread(Runnable r)
* Thread(Runnable r,String name)

Eg: class Multi extends Thread {

public void run() {

System.out.println("thread is running...");

}

public static void main(String args[]) {

Multi t1 = new Multi();

t1.start();

}

}

Multi class's defalut constructor is invoked from where Thread class constructor is invoked(by super()).

So Multi class object is thread object now.

**Runnable interface:**

* The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
* Runnable interface have only one method named run().
* public void run(): is used to perform action for a thread.

Eg: class Multi3 implements Runnable {

public void run() {

System.out.println("thread is running...");

}

public static void main(String args[]) {

Multi3 m1 = new Multi3();

Thread t1 = new Thread(m1);

t1.start();

}

}

If we are not extending the Thread class, the class object would not be treated as a thread object. So we need to explicitly create Thread class object and pass the object of the class that implements Runnable so that the run() method may be execute.

**Starting a thread:**

start() method of Thread class is used to start a newly created thread. It performs following tasks:

* A new thread starts(with new call-stack).
* The thread moves from New state to the Runnable state.
* When the thread gets a chance to execute, its target run() method will run.

**Thread Scheduler in Java**

* Thread scheduler in java is the part of the JVM that decides which thread should run.
* There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
* Only one thread at a time can run in a single process.
* The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

**Difference between preemptive scheduling and time slicing**

* Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence.
* Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

**Thread class methods:**

* **sleep()** method of Thread class is used to sleep a thread for the specified amount of time.
* **join()** method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.
* Yield() - Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
* **getName()** - returns the name of the thread
* **setName(String)** – change the specified name for the thread
* **getId()** - returns the id of the thread
* **currentThread()** method returns a reference to the currently executing thread object
* **getPriority()** - returns the priority of the thread
* **setPriority(int)** – changes the priority if the thread
* **setDaemon(boolean status)**
* **isDaemon()**

**Daemon Thread in Java:**

* It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
* Its life depends on user threads.
* It is a low priority thread.

**Java Daemon Thread examples:**

There are many java daemon threads running automatically

* gc
* finalizer

**Methods for Java Daemon thread by Thread class:**

* public void **setDaemon(boolean status)** - is used to mark the current thread as daemon thread or user thread.
* public boolean **isDaemon()** - is used to check that current is daemon.

**Note:** If you want to make a user thread as Daemon, it must not be started otherwise it will throw **IllegalThreadStateException**.

**Java Thread Pool**

* Java Thread pool represents a group of worker threads.
* In a thread pool, a group of fixed size threads are created.
* A thread from the thread pool is pulled out and assigned a job by the service provider.
* After completion of the job, thread is contained in the thread pool again.

**Fixed thread pool**

* One common type of thread pool is the fixed thread pool.
* This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread.
* Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

**Example scenario:**

Consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system.

With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

**A simple way to create a fixed thread pool:** is to invoke the **newFixedThreadPool** factory method in java.util.concurrent.Executors This class also provides the following factory methods:

* The **newCachedThreadPool** method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
* The **newSingleThreadExecutor** method creates an executor that executes a single task at a time.
* Several factory methods are **ScheduledExecutorService** versions of the above executors.

Eg: ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads

for (int i = 0; i < 10; i++) {

Runnable worker = new WorkerThread("" + i);

executor.execute(worker);//calling execute method of ExecutorService

}

executor.shutdown();

}

**Shutdown Hook:**

* If you want to execute some code before JVM shuts down, use shutdown hook.
* The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly.
* This can be achieved with addShutdownHook(Runnable r) method
* The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine.

Eg: Runtime r=Runtime.getRuntime();

r.addShutdownHook(new MyThread());

or

Runtime r = Runtime.getRuntime();

r.addShutdownHook(new Runnable() {

public void run() {

System.out.println("shut down hook task completed..");

}

});

**Deadlock**

* Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.
* Since, both threads are waiting for each other to release the lock, the condition is called deadlock.
* To avoid this problem thread synchronization is used.

**Thread Synchronization**

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

* Mutual Exclusive
  + Synchronized method.
  + Synchronized block.
  + static synchronization.
* Cooperation (Inter-thread communication in java)

**Synchronized method**

* If you declare any method as synchronized, it is known as synchronized method.
* Synchronized method is used to lock an object for any shared resource.
* When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Eg: synchronized void printTable(int n){}

**Synchronized block**

* Synchronized block can be used to perform synchronization on any specific resource of the method.
* Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
* If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Eg: synchronized (object reference expression) {

//code block

}

**Static synchronization**

If you make any static method as synchronized, the lock will be on the class not on object.

**Inter-thread communication**

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused which is running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

It is implemented by following methods of Object class:

* **wait()** - Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
* **notify()** - Wakes up a single thread that is waiting on this object's monitor. If many threads are waiting on this object, one of them is chosen to be awakened.
* **notifyAll()** - Wakes up all threads that are waiting on this object's monitor.

**Example scenario:**

* Threads enter to acquire lock.
* Lock is acquired by on thread.
* Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
* If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
* Now thread is available to acquire lock.
* After completion of the task, thread releases the lock and exits the monitor state of the object.

**Interrupting a Thread:**

* If any thread is in sleeping or waiting state 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.

3 methods are provided by the Thread class for interrupting a thread

* public void **interrupt()** - interrupts the thread
* public static boolean **interrupted()** - returns the interrupted flag either true or false
* public boolean **isInterrupted()** - returns the interrupted flag after seting the flag to false if it is true

**Reentrant Monitor in Java**

* Java monitors are reentrant means java thread can reuse the same monitor for different synchronized methods if another synchronized method is called from the method.
* Main advantage of reentrant is that it eliminates the possibility of single thread deadlocking.

Eg: public synchronized void m() {

n();

System.out.println("this is m() method");

}

public synchronized void n() {

System.out.println("this is n() method");

}