Multithreading in Java

- Amal Prasad

Introduction

- Thread is basically a lightweight sub-process, a smallest unit of processing
- **Multithreading in java** is a process of executing multiple threads simultaneously
- Multiprocessing and multithreading, both are used to achieve multitasking
- But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process
- Java Multithreading is mostly used in games, animation etc.

Advantages

- It **doesn't block the user** because threads are independent and you can perform multiple operations at same time
- You **can perform many operations together so it saves time**
- Threads are **independent** so it doesn't affect other threads if exception occur in a single thread

Amal Prasad

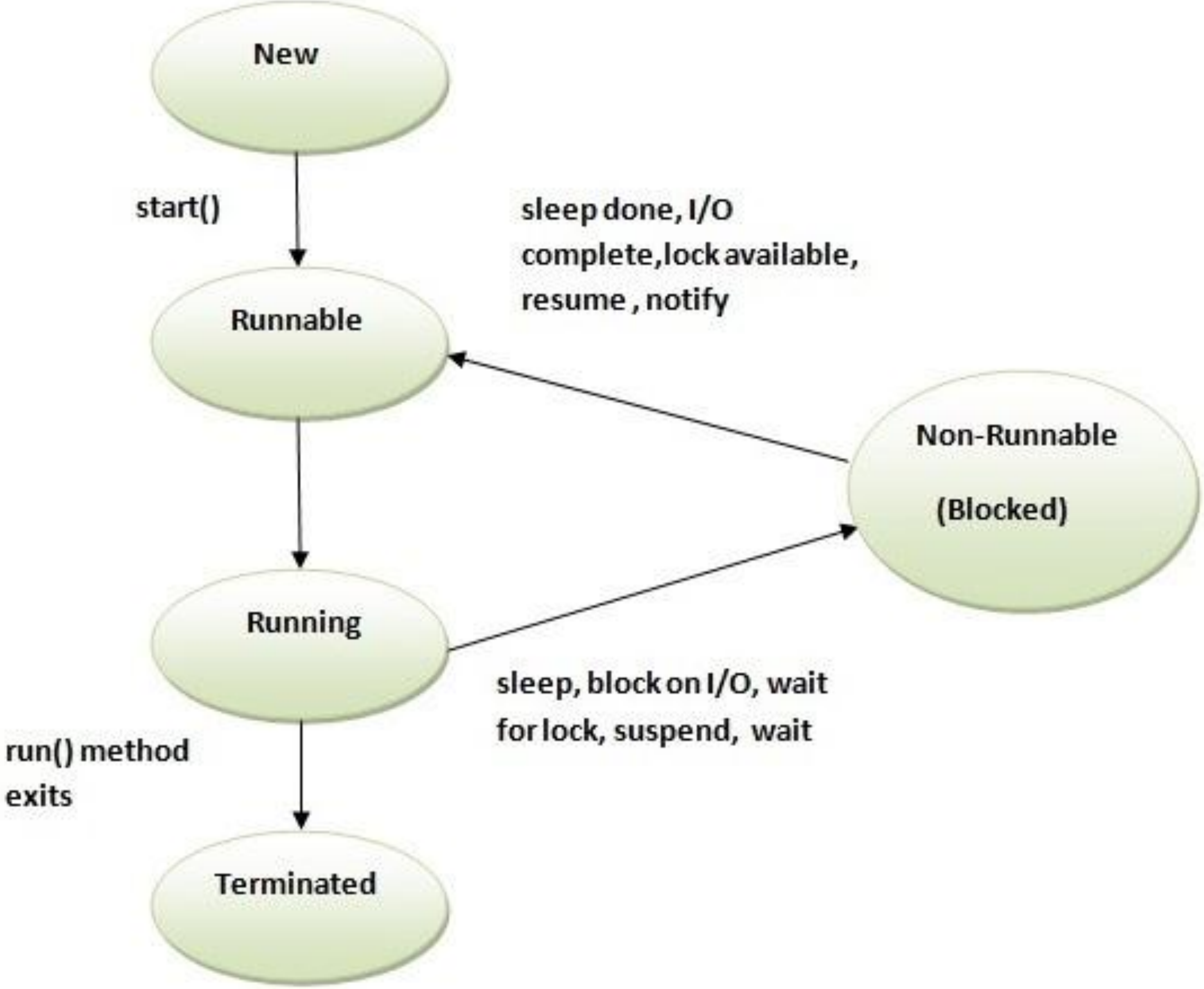
Multitasking

- Multitasking is a process of executing multiple tasks simultaneously
- We use multitasking to utilize the CPU
- Multitasking can be achieved by two ways:
    - Process-based Multitasking(Multiprocessing)
        - Each process have its own address in memory i.e. each process allocates separate memory area
        - Process is heavyweight
        - Cost of communication between the process is high
        - Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.
    - Thread-based Multitasking(Multithreading)
        - Threads share the same address space
        - Thread is lightweight
        - Cost of communication between the thread is low
- **At least one process is required for each thread**

Life cycle of a Thread (Thread States)

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

Thread States…



Amal Prasad

Create Thread

- There are two ways to create a thread:
  - By extending Thread class
    - Provide constructors and methods to create and perform operations on a thread
    - Extends Object class and implements Runnable interface
  - By implementing 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()

By extending Thread class

```
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();
 }
}
```

Implementing the Runnable interface

```
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();
 }
}
```

Thread Scheduler

- **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 pre-emptive or time slicing scheduling to schedule the threads
- **Difference between pre-emptive scheduling and time slicing**
- Under pre-emptive 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 re-enters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Thread Sleep

- sleep() method of Thread class is used to sleep a thread for the specified amount of time
- A a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on
- Example of Thread Sleep

Can we start a thread twice?

- No
- After starting a thread, it can never be started again
- Doing so, an *IllegalThreadStateException* is thrown

```
public class TestThreadTwice1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

What if we call run() method directly instead start() method?

- Each thread starts in a separate call stack
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack
- ***Problem if you direct call run() method***
  - there is no context-switching because here t1 and t2 will be treated as normal object not thread object

Amal Prasad

The join() method

- The 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
  - ***Example of join() method***
  - ***Example of join(long milliseconds) method***
    - when t1 is completes its task for 1500 milliseconds(3 times) then t2 and t3 starts executing

getName(), setName(String) and getId()

- Example
- **currentThread() method**
  - currentThread() method returns a reference to the currently executing thread object
  - ***Example of currentThread() method***

Amal Prasad

Naming a thread

```
class TestMultiNaming1 extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestMultiNaming1 t1=new TestMultiNaming1();
  TestMultiNaming1 t2=new TestMultiNaming1();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());

  t1.start();
  t2.start();

  t1.setName(“My New Name");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

Thread Priority

- Each thread have a priority
- Priorities are represented by a number between 1 and 10
- Thread scheduler schedules the threads according to their priority (known as pre-emptive scheduling)
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses
- **3 constants defiend in Thread class:**
  - public static int MIN_PRIORITY (1)
  - public static int NORM_PRIORITY (5)
  - public static int MAX_PRIORITY (10)
- **Example of priority of a Thread**

Amal Prasad

Daemon Thread in Java

- **Daemon thread in java** is a service provider thread that provides services to the user thread
- Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically
- Ex. Java daemon threads running automatically e.g. gc, finalizer etc.
- It provides services to user threads for background supporting tasks
- It is a low priority thread
- The java.lang Thread class provides two methods for java daemon thread

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

Daemon thread …

- Simple example of Daemon thread

Thread Pool

- **Java Thread pool** represents a group of worker threads that are waiting for the job and reuse many times
- In case of 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
- **Advantage of Java Thread Pool**
  - **Better performance** It saves time because there is no need to create new thread
- **Real time usage**
  - It is used in Servlet and JSP where container creates a thread pool to process the request
- **Example of Java Thread Pool**

Garbage Collection

- Garbage means unreferenced objects
- Garbage Collection is a way to destroy the unused objects
- **Advantage of Garbage Collection**
  - It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory
  - It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts
- **How can an object be unreferenced?**
  - By nulling the reference
    - Employee e=new Employee(); e=null;
  - By assigning a reference to another
    - Employee e1=new Employee();
    - Employee e2=new Employee();
    - e1=e2;//now the first object referred by e1 is available for garbage col lection
  - By anonymous object
    - new Employee();

Amal Prasad

finalize() & gc() method

- The finalize() method is invoked each time before the object is garbage collected
- Can be used to perform clean-up processing
- Defined in Object class
- **The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform clean-up processing**
- The gc() method is used to invoke the garbage collector to perform clean-up processing
- Found in System and Runtime classes
- Example of garbage collection in java

Thanks…