INTRODUCTION TO MULTITHREADING IN JAVA WITH EXAMPLE

Processes vs Threads

Processes are heavyweight tasks that require their own separate address spaces. Inter-process communication is expensive and limited. Context switching from one process to another is also costly.

Threads share the same address space and the same process. Threads require less overhead than processes. Inter-thread communication is inexpensive, and context switching from one thread to next is also of low cost.

Multitasking vs Multithreading

There are two types of **multitasking**: process-based (usually referred to simply as multitasking) and thread-based (usually referred to as multithreading).

In **process based multitasking**, the computer runs two or more programs (or processes) concurrently (e.g. a notepad and another program). Here, a program is the smallest unit of code that can be dispatched by the scheduler.

In **thread-based multitasking (or multithreading)**, a single program performs two or more tasks simultaneously through two more threads. Each thread defines a separate path of execution that can run in parallel. Here, the thread is the smallest unit of code that can be dispatched.

Thread states

A thread can be in one of the different states: New, Runnable, Running, Blocked and Terminated/Dead.

- When a thread object is created (e.g. Thread t = new Thread();) it is in **New state**.
- Once you invoke the threadObject's start() method (e.g. t.start()), the new thread will become **runnable**. Runnable threads are ready to run, but not running. After a suspended or blocking state also, a thread will become runnable.
- If there are many threads in the runnable state, thread scheduler will select one thread and make it**running**. A thread is in running state when it is currently executed by the processor. Only one thread per processor can be running at any particular time though we might not actually feel it.
- A thread can be suspended, sent to sleep or waiting, which causes the thread to go to **blocked state**. Once the blocked state is over, the thread goes to runnable state.
- At any time, a thread can be **terminated**, when the run() method gets completed or through an exception that halts its execution. Once terminated, a thread cannot be resumed and trying to do so will throw an IllegalStateException. This is called the **dead** state.

Creating Threads

You can create threads directly instantiating a Thread class or using the newer better and safer concurrency package features such as Executors.

Though it is not preferred to create threads directly, we will still see how we can create threads directly working with Thread class to understand basics well.

We can create a Thread class by extending the Thread class or implementing Runnable interface and passing it to a Thread class constructor. Both the Thread class and the Runnable interface contains a method called run; you have to override it. The run() method is called as a separate thread of execution when we call the start method. You can also call the run method directly, but then, it will be executed like any other method, not as a separate thread of execusion.

The Thread class also has other methods like init, start, stop etc., but we rarely need to override them and usually override only the run method. Unless you have a reason, like override any of those extra methods, you should implement Runnable as it gives you more flexibility as, in Java, you can extend only one class, but you can implement any number of interfaces along with extending a class. The Thread class also defines several methods to query and manage a thread like getName(), getPriority(), isAlive(), join(), run(), sleep(),start() etc.

The signature of run() method is:

public void run(){…}

Example 1: Extending Thread class

```
public class MyThread extends Thread {
    public void run()

    {

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

        {

            System.out.println(i);

            try {

                Thread.sleep(500);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }
    }
}
```

Example 2: Implementing Runnable

```
public class MyRunnable implements Runnable {
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println(i);
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Creating a Thread object (New State)

In case of extending Thread, you can create a Thread object as:

```
Thread t = new MyThread();
```

In case of implementing Runnable interface, you can create a Thread object as:

```
Thread t = new Thread(new MyRunnable());
```

Here we pass the Runnable interface implementation on to the Thread constructor.

Next we will see how to start a thread of execution.

Starting a Thread of Execution

A thread of execution can be started by calling start() on a thread object as:

```
t.start();
```

When you call start() method on a thread object, its run() method will be executed as a new thread.

You can even call the run method directly, though not recommended, this will not run as a separate thread, but just like any other method call from the calling thread as part of the calling thread.

Completing thread example

To demonstrate separate thread of execuion in parallel, we will create three thread objects and then start these three thread objects as separate path of execution. We will write a simple loop that prints from 1 to 10 and we can see that these three loops are executed in parallel, not serial in the order we called start.

Note however that, though these are three separate threads and can execute in parallel, it is upto the scheduler to decide which thread to run when these three are in runnable state; and it can even run them one by one if all of them are in runnable state. So we will send each thread to a small blocked state using Thread.sleep() so that the scheduler will have to then select from other two threads to execute. The static method Thread.sleep() causes the thread to suspend execution for a specified time. Since Thread.sleep can throw InterruptedException, you need to handle it using a try catch.

We will use the Thread classes created previously:

```
public class SimpleThreadExample {

    public static void main(String[] args) {

        MyThread t1 = new MyThread();

        MyThread t2 = new MyThread();

        Thread t3 = new Thread(new MyRunnable());

        t1.start();

        t2.start();

        t3.start();

    }

}
```

Try the example and understand it well, before moving any further learning about threads.

Daemon Threads

A daemon thread is a background thread and dies when the thread that created it ends. A thread that is not daemon is called a user thread. We can make a thread as daemon thread by calling setDaemon (true) on the thread instance. We can call setDaemon() for a thread only before it starts, else an IllegalThreadStateException will be thrown.

Source : http://javajee.com/introduction-to-multithreading-in-java-with-example