International Journal of Current Engineering and Technology
E-ISSN 2277 – 4106, P-ISSN 2347 - 5161
©2014 INPRESSCO®, All Rights Reserved
Available at http://inpressco.com/category/ijcet

General Article

Java Special Feature: Multithreading

Sangeeta Rani[A*] and Neetu Dhingra[A]

[A]BLS Institute of Technology Management, Bahadurgarh- 124507, Haryana, India

Accepted 30 Nov 2014, Available online 01 Dec 2014, **Vol.4, No.6 (Dec 2014)**

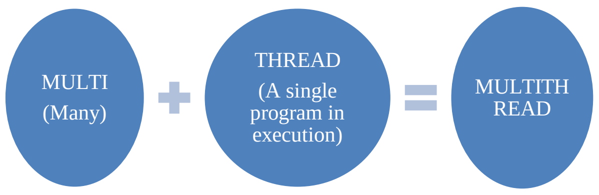
Abstract

Multithreading is the most crucial feature of the java. In real world parallel processing is needed to perform multiple tasks at the same time for the purpose of resource management and time management. This goal generates an urgent need for multithreading. In this paper first, we show the importance of multithreading in java and then implementation of threads in java.

Keywords: *Thread, Runnable, Interface, Start, Priority, Life Cycle, Run, Method*

Introduction

Multithreading means "to run multiple threads at the same time"



Multithreading concept probably similar to the *multitasking:* the ability to have more than one program working at what seems like the same time. For example you are playing songs or working in notepad at the same time. Of course, unless you have a multiple-processor machine, what is really going on is that the operating system is doing out resources to each program, giving the impression of parallel activity. This resource distribution is possible because while you may think you are keeping the computer busy by, for example, entering data, most of the CPU's time will be idle.

Multitasking can be done in two ways, depending on whether the operating system interrupts programs without consulting with them first, or whether programs are only interrupted when they are willing to yield control. The former is called *preemptive multitasking;* the latter is called *cooperative* (or, simply, non preemptive). *Multitasking.* Windows 3.1 and Mac OS 9 are cooperative multitasking systems, and UNIX/Linux, Windows NT, and OS X are preemptive.

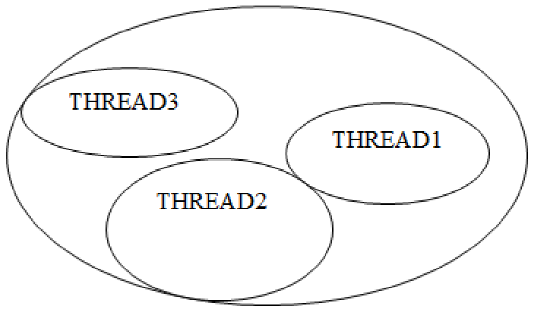
Multithreaded programs extend the idea of multitasking by taking it one level lower: individual programs will appear to do multiple tasks at the same time. Each task is usually called a *thread*—which is short for thread of control. Programs that can run more than one thread at once are said to be *multithreaded.* Think of make it seem as though each thread has its own CPU—with registers, memory, and its own code. (Addison-Wesley, 1999).

*Corresponding author **Sangeeta Rani** and **Neetu Dhingra** are working as Assistant Professors.

Defining Threads

To understand multithreading, the concepts *process* and *thread* must be understood. A *process* is a program in execution. A process may be divided into a number of independent units known as *threads*.

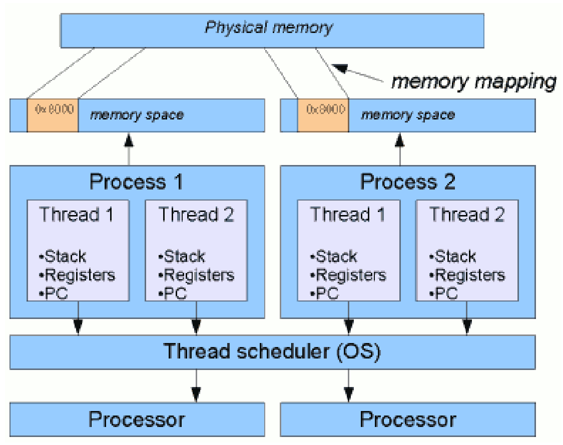
Thread:- A *thread* is a dispatchable unit of work. Threads are *light-weight* processes within a process
Process:- A process is a collection of one or more threads and associated system resources. The difference between a process and a thread is shown:



A process may have a number of threads in it. A thread may be assumed as a subset of a process. A process containing single and multiple threads. If two applications are run on a computer (MS Word, MS Powerpoint), two processes are created.

Multitasking of two or more processes is known as *process-based multitasking*. Multitasking of two or more threads is known as *thread-based multitasking*. The concept of multithreading in a programming language refers to thread-based multitasking. Process-based multitasking is totally controlled by the operating system. But thread-based multitasking can be controlled by the programmer to some extent in a program.

Switching between Threads



Lifecycle of Thread

The life cycle of threads in Java is very similar to the life cycle of processes running in an operating system. During its life cycle the thread moves from one state to another depending on the operation performed by it or performed on it as illustrated in Fig. 14.4. A Java thread can be in one of the following state
New State: A Thread is called in new state when it is created. To create a new thread you may create an instance of Thread class or you can create a subclass of Thread and then you can create an instance of your class.

Example

Thread thread = new Thread ();
Newly created thread is not in the running state. To move the thread in running state the start() method is used.

Runnable State: A Thread is called in runnable state when it is ready to run and its waiting to give control. To move control to another thread we can use yield() method.

Example

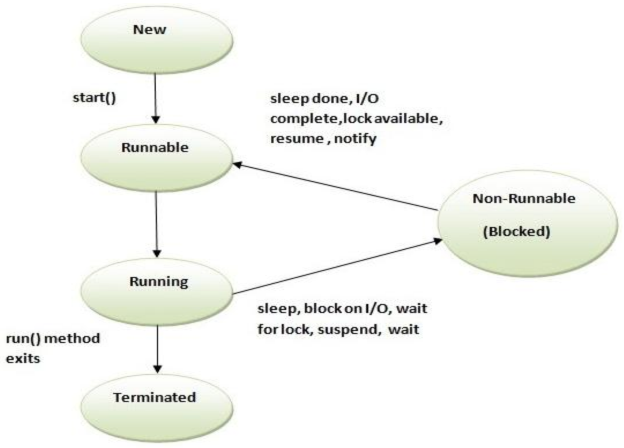
Thread thread = new Thread();
thread.yield();

Running State: A Thread is called in running state when it is in its execution mode. In this state control of CPU is given to the Thread. To execute a Thread the scheduler select the specified thread from runnable pool. **Blocked State:** A Thread is called in blocked state when it is not allowed to enter in runnable and/or running state. A thread is in blocked state when it is suspend, sleep or waiting.

Example

Thread thread = new Thread();
thread.sleep();

Dead/Terminated State: A Thread is called in dead state when its run() method execution is completed. The life cycle of Thread is ended after execution of run() method. A Thread is moved into dead state if it returns the run() method.


New
start()
Runnable
sleep done, I/O complete, lock available, resume, notify
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

Implementation of threads

There are two ways to implement threads in java

1. By extending Thread class.
2. By implementing Runnable interface in java.

To implement the thread we first introduce about some methods which are used in the implementation of threads.
1.START():- To execute any thread we use this method.
2.RUN():- It is the heart of any thread.only run() method is used to perform any task in the java.

By extending thread class

```
Class  A extends Thread
{
public void run()
{
System.out.println("welcome");
}
}
Class  B extends Thread
{
public void run()
{
System.out.println("IN");
}
}
Class  C extends Thread
{
public void run()
{
System.out.println("Java");
}
}
```

```
Class D
{
Public static void main(String a[])
{
A a1=new A();
B b1=new B();
C c1=new C()
a1.start();
b1.start();
c1.start();
}
}
```

By implementing Runnable Interface

```
Class  A implements Runnable
{
public void run()
{
System.out.println("welcome");
}
}
Class  B implements Runnable
{
public void run()
{
System.out.println("IN");
}
}
Class  C implements Runnable
{
public void run()
{
System.out.println("Java");
}
}
Class D
{
Public static void main(String a[])
{
A a1=new A();
B b1=new B();
C c1=new C()
Thread  t1=new Thread(a1);
Thread  t2=new Thread(b1);
Thread  t3=new Thread(c1);
t1.start();
t2.start();
t3.start();
}
}
```

Thread Priority

In Java, thread scheduler can use the thread priorities in the form of integer value to each of its thread to determine the execution schedule of threads . Thread gets the ready-to-run state according to their priorities. The thread scheduler provides the CPU time to thread of highest priority during ready-to-run state.

Priorities are integer values
1 (lowest priority given by the constant **Thread.MIN_PRIORITY)**
10 (highest priority given by the constant **Thread.MAX_PRIORITY**)
5 (default priority is **Thread.NORM_PRIORITY**)

Constant	Description
Thread.MIN_PRIORITY	The maximum priority of any thread (an int value of 10)
Thread.MAX_PRIORITY	The minimum priority of any thread (an int value of 1)
Thread.NORM_PRIORITY	The normal priority of any thread (an int value of 5)

The methods that are used to set the priority of thread shown as:

Method	Description
setPriority()	This is method is used to set the priority of thread.
getPriority()	This method is used to get the priority of thread.

When a Java thread is created, it inherits its priority from the thread that created it. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. In Java runtime system, preemptive scheduling algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new higher priority thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the round-robin algorithm is applied in which the scheduler chooses one of them to run according to their round of time-slice.

Thread Scheduler

Threading scheduler usually applies one of the two following strategies:

Preemptive scheduling? If the **new thread** has a higher priority than current running thread leaves the runnable state and higher priority thread enter to the runnable state.

Time-Sliced (Round-Robin) Scheduling? A running thread is allowed to be execute for the fixed time, after completion the time, current thread indicates to the another thread to enter it in the runnable state.
You can also set a thread's priority at any time after its creation using the set Priority method. Let's see, how to set and get the priority of a thread.

```
class MyThread1 extends Thread{
MyThread1(String s){
super(s);
  start();
  }
  public void run(){
  for(int i=0;i<3;i++){
  Thread cur=Thread.currentThread();
  cur.setPriority(Thread.MIN_PRIORITY);
  int p=cur.getPriority();
  System.out.println("Thread Name  :"+Thread.currentThr
ead().getName());
  System.out.println("Thread Priority  :"+cur);
  }
  }
}
  class MyThread2 extends Thread{
  MyThread2(String s){
  super(s);
  start();
  }

public void run(){
  for(int i=0;i<3;i++){
  Thread cur=Thread.currentThread();
  cur.setPriority(Thread.MAX_PRIORITY);
  int p=cur.getPriority();
  System.out.println("Thread Name  :"+Thread.currentThr
ead().getName());
  System.out.println("Thread Priority  :"+cur);
  }
  }
}
public class ThreadPriority{
  public static void main(String args[]){
  MyThread1 m1=new MyThread1("My Thread 1");
  MyThread2 m2=new MyThread2("My Thread 2");
  }
}
```

Output of the Program

```
C:\sangeeta>javac ThreadPriority.java

C:\sangeeta>java ThreadPriority
Thread Name :My Thread 1
Thread Name :My Thread 2
Thread Priority :Thread[My Thread 2,10,main]
Thread Name :My Thread 2
Thread Priority :Thread[My Thread 2,10,main]
Thread Name :My Thread 2
Thread Priority :Thread[My Thread 2,10,main]
Thread Priority :Thread[My Thread 1,1,main]
Thread Name :My Thread 1
Thread Priority :Thread[My Thread 1,1,main]
Thread Name :My Thread 1
Thread Priority :Thread[My Thread 1,1,main]
```

In this **program** two threads are created. We have set up maximum priority for the first thread "**MyThread2**" and minimum priority for the first thread "**MyThread1**" i.e. the after executing the program, the first thread is executed only once and the second thread "**MyThread2**" started to run until either it gets end or another thread of the equal priority gets ready to run state

C**onclusion**

In this paper, we have proposed life cycle for the thread .In this we implement the methods to implement the threads in java. And how we can get the priority to the thread that execute first.

The future scope of this paper to provide the synchronization between the threads for the concurrency control.

References

K. Arnold, J. Gosling (1998), The Java™ Programming Language (2nd edition), Addison-Wesley.

ANSI/IEEE IEEE Standard for Binary Floating-Point Arithmetic; ANSI/IEE Std. 754-1985; 1985.

Aonix(1997); ObjectAda Integrated Development Environment, 7(1)

K. Arnold (1996); The Java Programming Language, Professional Development Seminar delivered to Boston

B. Brosgol (1998); A Comparison of the Concurrency Features of Ada 95 and Java, Proc. SIGAda

T.A. Cargill (1993), The Case Against Multiple Inheritance in C++, in The Evolution of C++ (J. Waldo, ed.), The MIT Press pp. 101-110, The MIT Press

D. Flanagan (1997); Java in a Nutshell (2nd edition), O'Reilly & Associates

J. Gosling, B. Joy, G. Steele (1996); The JavaTM Language Specification; Addison Wesley

Unicode Consortium (1996); The Unicode Standard: Worlwide Character Encoding, Version 2.0; Addison-Wesley, 1996, ISBN 0-201-48345-

Herbert Schildt The Complete Reference, Seventh Edition.

J. Waldo (1993), The Case For Multiple Inheritance in C++, in The Evolution of C++ (J. Waldo, ed.) ,The MIT Press, pp. 111-120