

Multithreading

Introduction

Before diving right into the concept of multithreading let's have a look at some of the basic underlying concepts. We all know about a program. A **program** is a set of statements for solving a particular problem/task. Whenever a program is executed, the program transforms into a process. So, a running program is known as a **process**. Program is passive code and process is active code which is being executed by the CPU.

The difference between a program and process is: first, a program is not executed by the CPU while a process is the running program which is executed by the CPU. Second, a program does not perform any useful operations while a process performs a specified task. Third, a program does not need any system resources while a process requires resources like: CPU, memory (RAM), program counter, instruction register, CPU registers etc. That is why a process is **heavyweight**.

Now, what is a thread? A **thread** is defined as a light weight process. A thread is part of a program. Without a program there is no thread. All threads share the resources of a process and does not require as many resources as the process. That is why a thread is light weight.

Single Tasking

In computer science, a process or a thread can also called as a task or job. Single tasking refers that only one task can be executed at a time by the CPU. Example operating system for single tasking is DOS.

Multi Tasking

Multi tasking refers to the capability of the system to execute more than one task at a time. Example for multi tasking operating system is windows family. Multi tasking based on process is known as Multi Processing and multi tasking based on threads is known as Multi Threading.

Multi Processing

Multi processing refers to the capability of the system to execute more than one process at a time. For example in current generation computers an user able to play music while editing a word document.

Multi Threading

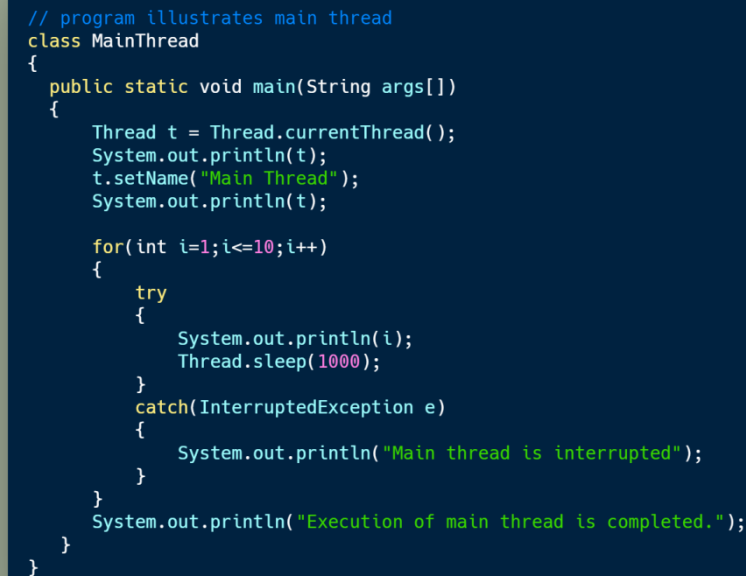
If a program is divided into several parts and all these parts are executed concurrently by the CPU, it is known as multi threading and each part is considered a thread. The advantage of multi threading is the increase in CPU's efficiency. In a single processor computer, even though the operating system supports multi processing and multi threading, the CPU never runs two or more processes or threads at a time. A single CPU can run only either one process or one thread at a time.

Multi Processor

This is related to the hardware rather than the software. Multi processor refers to multiple processors (CPUs). If there are multiple processors in the system, each processor can run a processor or a thread concurrently. So, in multi processor systems we can achieve true parallelism or concurrency.

The main thread

By default every java stand alone application will have a single thread namely the main thread. The programmer will not have any control over starting the main thread as it is in the hands of JVM. The programmer can suspend, resume and stop the main thread. The programmer can use the `currentThread()` method in the Thread class obtain the reference of the main thread. Let's consider the following program which demonstrates controlling the main thread:



```
// program illustrates main thread
class MainThread
{
    public static void main(String args[])
    {
        Thread t = Thread.currentThread();
        System.out.println(t);
        t.setName("Main Thread");
        System.out.println(t);

        for(int i=1;i<=10;i++)
        {
            try
            {
                System.out.println(i);
                Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {
                System.out.println("Main thread is interrupted");
            }
        }
        System.out.println("Execution of main thread is completed.");
    }
}
```

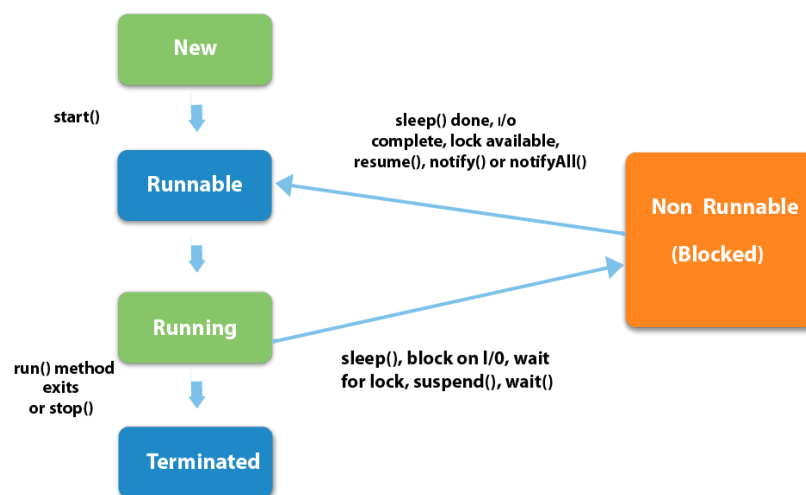
There are a few important points to remember regarding the main thread. They are:

1. The main thread is the parent thread for all other user defined threads.
2. The main thread is the last thread that is to be executed in a java program.

Thread life cycle

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated



1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it's code is yet to be run and hasn't started to execute.
2. **Runnable State:** A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.
3. **Blocked/Waiting state:** When a thread is temporarily inactive, then it's in one of the following states:
 1. Blocked
 2. Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states does not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the scheduler picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

4. **Running:** When thread is executing, it's state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it's state to Running. Then CPU starts

executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

5. **Terminated State:** A thread terminates because of either of the following reasons:
- Because it exists normally. This happens when the code of thread has entirely executed by the program.
 - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.
- A thread that lies in a terminated state does no longer consumes any cycles of CPU.

Creating threads in Java

A thread can be created in a java program either by using the predefined class **Thread** or the predefined interface **Runnable**. Both are available in the package **java.lang**.

1. Using Threadclass

Using the Thread class we can create threads. The necessary steps are as follows:

- a. Extend the **Thread**class.
- b. Override the runmethod.
- c. Call the startmethod.

The run() method is the entry point for every thread in java. This run method is invoked automatically when the start() method is called.

Some the methods in the Thread class are as shown below:

Method	Meaning
final String getName()	Obtains a thread's name.
final int getPriority()	Obtains a thread's priority.
final boolean isAlive()	Determines whether a thread is still running.
final void join()	Waits for a thread to terminate.
void run()	Entry point for the thread.
static void sleep(long milliseconds)	Suspends a thread for a specified period of milliseconds.
void start()	Starts a thread by calling its run() method.

//Program illustrating threads: Using thread class

```

class Hi extends Thread {
    public void run() {
        for(int i=1; i<=5; i++){
            System.out.println("Hi");
            try { Thread.sleep(500); } catch(Exception e) { }
        }
    }
}

class Hello extends Thread {
    public void run() {
        for(int i=1; i<=5; i++){
            System.out.println("Hello");
            try { Thread.sleep(500); } catch(Exception e) { }
        }
    }
}

class ThreadDemo{
    public static void main(String args[]){
        Hi t1 = new Hi();
    }
}

```

```

        Hello t2 = new Hello();

        t1.start();
        t2.start();
        System.out.println("isAlive(t1)" + t1.isAlive());
        try{
            t1.join();
            t2.join();
        }catch(Exception e){ }
        System.out.println("isAlive(t1)" + t1.isAlive());
        System.out.println("Bye");
    }
}

```

2. Using Runnableinterface

The interface **Runnable** is a predefined interface available in the package java.lang. The method available in the Runnable interface is the run method. So, the class which implements Runnable interface must provide the implementation for the run method. The necessary steps for creating a thread using the Runnable interface are as follows:

1. Implement the Runnableinterface.
2. Override the run()method.
3. Call the start()method.

//Program illustrating threads : using runnable interface

```

class Hi implements Runnable {
    public void run() {
        for(int i=1; i<=5; i++){
            System.out.println("Hi");
            try { Thread.sleep(500); } catch(Exception e) { }
        }
    }
}

class Hello implements Runnable {
    public void run() {
        for(int i=1; i<=5; i++){
            System.out.println("Hello");
            try { Thread.sleep(500); } catch(Exception e) { }
        }
    }
}

class RunnableDemo{
    public static void main(String args[]){
        Runnable obj1 = new Hi();
        Hello obj2 = new Hello();
        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);
        t1.start();
        t2.start();

    }
}

```

Which one to choose for creating threads among Thread and Runnable?

As we know that there are two ways for creating a thread either using the Thread class or using the Runnable interface, the question arises which method to use? The programmer can use the Thread class if he/she knows that the class will not extend any other classes other than the Thread class. Otherwise, if the class has to extend another class, he/she can go with Runnable interface.

Thread Priorities

When threads are created, the default priority for each thread is 5. The priority of a thread signifies the importance given to the thread by the CPU. If the thread is having high priority, it gets more time to be executed by the CPU and vice versa. The priority levels in java are from 1 to 10. A thread with priority 1 is said to have the least priority and a thread with priority 10 is said to have the highest priority. There are two methods to handle priorities for threads. They are:

- `void setPriority(int prioritylevel)` – To assign priority to a thread
- `int getPriority()` – To retrieve the priority of a thread.

Instead of using numbers for priorities, we can use symbolic constants.

`MIN_PRIORITY` for 1, `NORM_PRIORITY` for 5 and `MAX_PRIORITY` for 10.

```
//Program illustrating thread priority
class Hi extends Thread {
    Hi() { this.start(); }
    public void run() {
        for(int i=1; i<=5000; i++){
            System.out.print("Hi ");
        }
    }
}

class Hello extends Thread {
    public void run() {
        for(int i=1; i<=5000; i++){
            System.out.print("Hello ");
        }
    }
}

class ThreadPriorityDemo {
    public static void main(String args[]){
        Hello t2 = new Hello();
        Thread t1 = new Hi();

        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(10);
        System.out.println("Priority of t1 : "+ t1.getPriority());
        System.out.println("Priority of t2 : "+ t2.getPriority());
        t2.start();
    }
}
```

Thread Synchronization

We all know that threads in a program execute concurrently or in parallel to one another. There will be no problem as long as the threads work on different resources. But, if multiple threads work on the same resource such as a method, file or a database, inconsistencies might arise. For example consider the following scenario: Imagine a printer is being accessed at the same time from two computers across a LAN. Also the two computers are trying to print files at the same time. What will happen in this case? Will the printer print the two files at the same time? The answer is no. The printer with the help of OS maintains a queue to stop the parallel requests and processes them one after another. This helps the printer to produce consistent output.

The process or mechanism which allows to maintain consistency when a resource is being accessed by multiple threads is known as synchronization. Java supports synchronization of

threads by providing:

1. synchronized methods
2. synchronized statement

Synchronized methods

When multiple threads are accessing the same method, we can synchronize the method with the help of **synchronized** keyword. The syntax of a synchronized method will be as shown below:

```
synchronized returntype methodname(parameters...)
{
    Statements(s);
}
```

Example:

```
class Counter
{
    int count;
    public synchronized void increment()
    {
        count++;
    }
}

class CounterThread extends Thread
{
    Counter c1;
    CounterThread(Counter c1)
    {
        this.c1=c1;
    }
    public void run()
    {
        for(int i=0; i<1000; i++)
            c1.increment();
    }
}

class SyncDemo{
    public static void main(String args[]){
        Counter c = new Counter();
        CounterThread t1 = new CounterThread(c);
        CounterThread t2 = new CounterThread(c);
        CounterThread t3 = new CounterThread(c);
        t1.start(); t2.start(); t3.start();
        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (Exception e) { }
        System.out.println("Count : " + c.count);
    }
}
```

Synchronized Statement

When the methods are developed by another developer or when the programmer has no access to the method definition, in such cases we can synchronize the call to the method by using the **synchronized statement**. The syntax of synchronized statement is as shown below:

```
synchronized(object)
```

```

{
    Statement(s);
}

```

Example:

```

class Caller implements Runnable
{
    Thread t;
    CallMe target;
    String msg;
    Caller(CallMe tar, String str)
    {
        target = tar; msg = str;
        t = new Thread(this); t.start();
    }
    public void run()
    {
        synchronized(target)
        {
            target.call(msg);
        }
    }
}
class CallMe
{
    void call(String msg)
    {
        System.out.print "["+msg);
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Thread is interrupted");
        }
        System.out.println("]");
    }
}
class SyncDemo
{
    public static void main(String args[])
    {
        CallMe target = new CallMe();

        Caller one = new Caller(target, "Welcome");
        Caller two = new Caller(target, "to");
        Caller three = new Caller(target, "Java");

    }
}

```

Daemon Threads

A daemon thread is a thread which runs in the background continuously. In java, threads can be divided into two categories:

- 1) User threads and
- 2) Daemon threads.

The user threads are the threads which are created by the programmer. Daemon threads are those threads which are initially user threads but are converted into daemon threads.

When the program has atleast one user thread under execution, JVM cannot terminate that process.

But, when the program has no user threads and still has daemon threads under execution, JVM can terminate that process. So, be careful regarding what code you are placing in the daemon thread. To work with daemon threads there are two predefined methods in the Thread class:

void setDaemon(boolean flag) – To convert a user thread into daemonthread

boolean isDaemon() – To know whether a thread is daemon thread or not.

Example:

```
class MyThread extends Thread
{
    public void run()
    {
        try{Thread.sleep(2000);}catch(Exception e){ }
        System.out.println("My name is " + this.getName());
    }
}
class DaemonDriver
{
    public static void main(String args[])
    {
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        t1.setDaemon(true);
        t3.setDaemon(true);
        t1.start(); t2.start(); t3.start();
        System.out.println("Is t1 a Daemon : " + t1.isDaemon());
        System.out.println("Is t2 a Daemon : " + t2.isDaemon());
        System.out.println("Is t3 a Daemon : " + t3.isDaemon());
    }
}
```

Thread Groups

We can control threads not only individually but also in groups. A thread group is a collection of threads. A thread group allows the user to control threads more effectively. To work with thread groups, java provides a predefined class **ThreadGroup** which is available in the package **java.lang**. To create a thread group we can use the following constructors of the ThreadGroup class:

1. ThreadGroup(Stringgroupname)
2. ThreadGroup(ThreadGroup parent, Stringgroupname)

After creating a thread group, we can add threads to the thread group by using the following constructors of the **Thread** class:

1. Thread(ThreadGroup ref, Runnableobject)
2. Thread(ThreadGroup ref, Stringthreadname)
3. Thread(ThreadGroup ref, Runnable object, String threadname)

Following are some of the methods available in the

ThreadGroupclass:

1. getName() – To retrieve the name of the threadgroup.
2. setMaxPriority() – To set the maximum priority for all the threads in the threadgroup
3. getMaxPriority() – To get the maximum priority for all the threads in the threadgroup
4. start() – To start all the threads in thegroup
5. stop() – To stop all the threads in thegroup
6. suspend() – To suspend all the threads in thegroup
7. resume() – To resume all the threads in thegroup
8. list() – To print out information regarding the thread group and the threads in thegroup.

Consider the following program which demonstrates thread groups:

```
//Program illustrate inter thread communication
class ChildThread implements Runnable
{
    Thread t;
    ChildThread(ThreadGroup a, String name)
    {
        t = new Thread(a,this,name);
        a.setMaxPriority(5);
        System.out.println(t.getName()+"belongs to the group: "+a.getName());
        System.out.println("Maximum priority of "+a.getName()+" is:"+a.getMaxPriority());
        t.start();
    }
    public void run()
    {
        try
        {
            for(int i=1;i<=10;i++)
            {
                System.out.println(t.getName()+" "+i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println(t.getName()+" is interrupted");
        }
    }
}

class ThreadGroups
{
    public static void main(String args[])
    {
        ThreadGroup tgl = new ThreadGroup("Group A");
        ChildThread one = new ChildThread(tgl, "First Thread");
        ChildThread two = new ChildThread(tgl, "Second Thread"); tgl.list();
        try
        {
            Thread.sleep(3000);
            System.out.println("All the threads in the thread group, Group A will be stopped now.");
            tgl.stop();
        }
        catch (InterruptedException e)
        {
            System.out.println("main thread is interrupted");
        }
    }
}
```