Articles » Languages » Java » General

Java Thread Tutorial

By **Arash** M. Dehghani, 8 Sep 2013

Introduction

In programming, sometimes you need to do some tasks together **(simultaneously)**, for example copying a file and showing the progress, or playing a music while showing some pictures as a slideshow, or a listener handles cancel event for aborting a file copy operation. If a program needs to run (do) more than one task **(thread)** at a time, that kind of a program is called a **concurrent (multi-thread)** program.

The main problem with a multi-threaded program is **synchronization** between the running threads, because each thread wants to finish its process ASAP, for example, a program with three threads where each thread has a for loop inside it, and each iteration prints A, B, C. By default, there is not any warranty they will finish their task at the same time because they started at the same time, maybe the third thread will finish its work before the first and second threads.

Synchronizing threads is not such a complex or hard work, it could be done very easily. In some cases it may need a lot of code and might complicate the application flow.

Table of Contents

Why Concurrent Programs?

Let's answer this question with another question, why single-threaded programs? In an application sometimes some tasks need to run together, **specially in a graphical user interface**, a game as an example, you have to track time, manage the objects, play the sounds at the same time, and this could be done in a multi-threaded manner. Even a simple program like calling an internet address (resource), user should be able to cancel the operation, and it could be done in two separate threads. In fact we have to separate a program into small units that would work concurrently, but sometimes it's impossible for some sort of applications (like a sequential math formula, or connecting to a database).

How to convert an application to become a multi-threaded application? Is it possible?

It is impossible for programs that each step is completely dependent to the previous step, and usually does just one task overall, a timer program for instance.

We need to find tasks in an application that need to or would run with another thread, for example in a game. We have to run the loading screen as a thread to show the user the progress of game loading which is running by loading modules together, or we **would** run the file loading task as a thread to avoid screen

freeze by the GUI thread.

- If a program performs more than one task (usually GUI apps), it is possible to run each task as a thread, **if** and only if there are no tight step-by-step dependencies belonging to the tasks
- If a task could be split to small units of work, **if** and only if there is no line-by-line dependencies in the tasks, for example initializing a big array

Note that in concurrent applications, one of the purposes is to get the maximum power of the processor, either GPU or CPU, if a file conversion utilizes 10% of CPU, why don't we run 10 concurrent threads to utilize 100% of CPU and reduce the execution time by 10 times?!

Creating Threads

So first of all, we need to know how to create a thread in our application. In the Java language a thread is specified with a `Runnable` interface or a `Thread` class. In C/C++ (before 11) there is no certain impl of thread, some libs are available, I suggest **pthread**, it's easy and has everything a programmer needs. Here we use Java, but you could do it with any language, any library, you just need to know the logic.

Java

In the Java language, as mentioned, either the `Runnable` interface or the `Thread` class is used for specifying a thread. The `Thread` class starts the thread, it gets an implementation of `Runnable` and starts the thread either in a separate thread or the current thread. The `Runnable` interface just has a `void` method called `Run` which does not accept any arguments, so for any input, you would pass it via the class that has just implemented it.

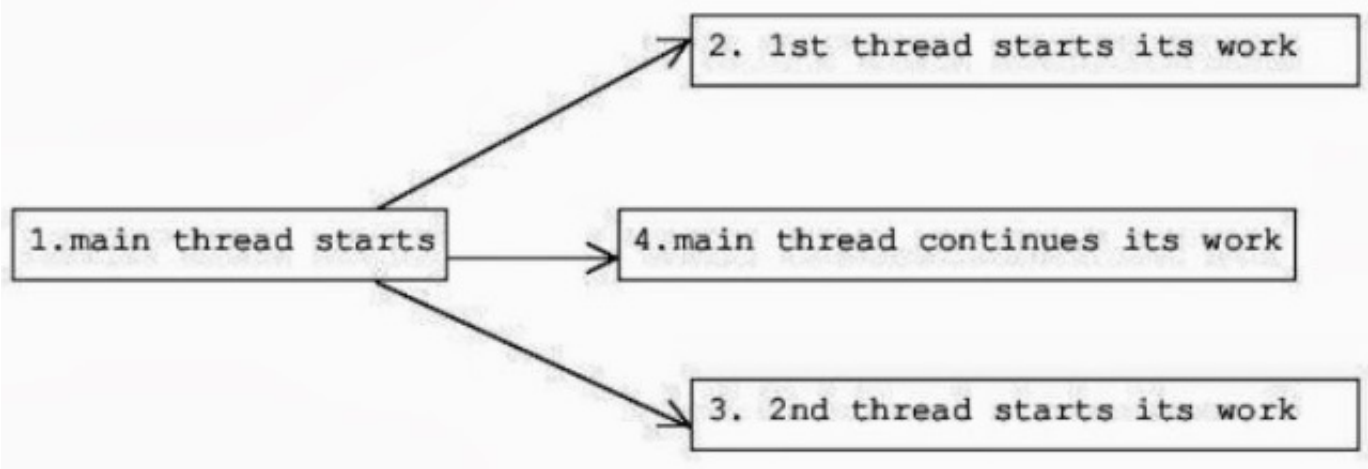
Example

The following example is just about two threads that get started from the main thread (first thread).

```
package arash.blogger.example.thread;
/**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
 public static void main(String[] args) {
  Runnable r0,r1;//pointers to thread methods
  r0=new FirstIteration(); //init Runnable
  r1=new SecondIteration();
  Thread t0,t1;//Thread class is used for starting a thread (runnable instance)
  t0=new Thread(r0);//init thread object, but haven't started yet
  t1=new Thread(r1);
  t0.start();//start the thread simultaneously
  t1.start();
  System.out.print("Threads started, no surprise here!\n");
 }
}
class FirstIteration implements Runnable{
 @Override
 public void run() {//thread starts from here
  for(int i=0;i<20;i++){
   System.out.print("Hello from 1st. thread, iteration="+i+"\n");
  }
```

```
 }
}
class SecondIteration implements Runnable{
 @Override
 public void run() {
  for(int i=0;i<20;i++){
   System.out.print("who just called 2st. thread? iteration="+i+"\n");
  }
 }
}
```

Just try the above code many times and you will be faced with **different responses**, this is because threads have **Race condition**s, which means each thread wants to finish its job **ASAP** (it does not belong to JVM). The above example is about two threads without any synchronization. The following diagram shows how the application does that.



Exercises

And now, this is your turn, just stop using Ctrl+c and Ctrl+v, and try to do the following practices (feel free to contact for any issues).

- Develop a program like the above example, start with two threads, then the concrete threads will create two threads with their thread.
- A program that creates 100 randomized name folders concurrently

For better understanding of "why should I write it multi-threaded?" just develop each one in a single-thread too, and have a comparison.

Passing argument(s) to threads

It's really easy to pass a value to a thread, but it is impossible with the `Runnable` implicitly, it could be done by host class, constructor, setters, etc...

Example

Check out the following example for a better understanding.

```
package arash.blogger.example.thread;
/**
```

```
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
 public static void main(String[] args) {
  Runnable r0,r1;//pointers to a thread method
  r0=new FirstIteration("Danial"); //init Runnable, and pass arg to thread 1
  SecondIteration si=new SecondIteration();
  si.setArg("Pedram");// pass arg to thread 2
  r1=si;
  Thread t0,t1;
  t0=new Thread(r0);
  t1=new Thread(r1);
  t0.start();
  t1.start();
  System.out.print("Threads started with args, nothing more!\n");
 }
}
class FirstIteration implements Runnable{
 public FirstIteration(String arg){
 //input arg for this class, but in fact input arg for this thread.
  this.arg=arg;
 }
 private String arg;
 @Override
 public void run() {
  for(int i=0;i<20;i++){
   System.out.print("Hello from 1st. thread, my name is "+
     arg+"\n");//using the passed(arg) value
  }
 }
}
class SecondIteration implements Runnable{
 public void setArg(String arg){//pass arg either by constructors or methods.
  this.arg=arg;
 }
 String arg;
 @Override
 public void run() {
  for(int i=0;i<20;i++){
   System.out.print("2)my arg is="+arg+", "+i+"\n");
  }
 }
}
```

In the above example we passed a `String` value as an argument for threads, but remember that it should get passed by the hosted class (impl class).

Exercises

And now, here is your turn, try to do the following:

- An application which multiply two float arrays A, B and store the results in the array C (c[0]=a[0]*b[0])
- A program that gets string line(s) from the user and saves them into 10 files
- A program that creates 100 folders named from 0 to 99 concurrently
- A program that kills every other OS process concurrently (**Dangerous**!, don't really try to do that)

Managing Threads

Well, we have just done about creating and running threads, but how do we manage them? How do we sleep (idle) a thread (stop for a particular time) at a point? How do we pause (stop working till a resume signal) a thread? How do we shutdown a running thread? Or how do we get the current running thread?

Thread's Properties

Generally in every language and platform, threads have some common properties, for example, **thread id** or **private memory**.

Properties

- **Thread id**

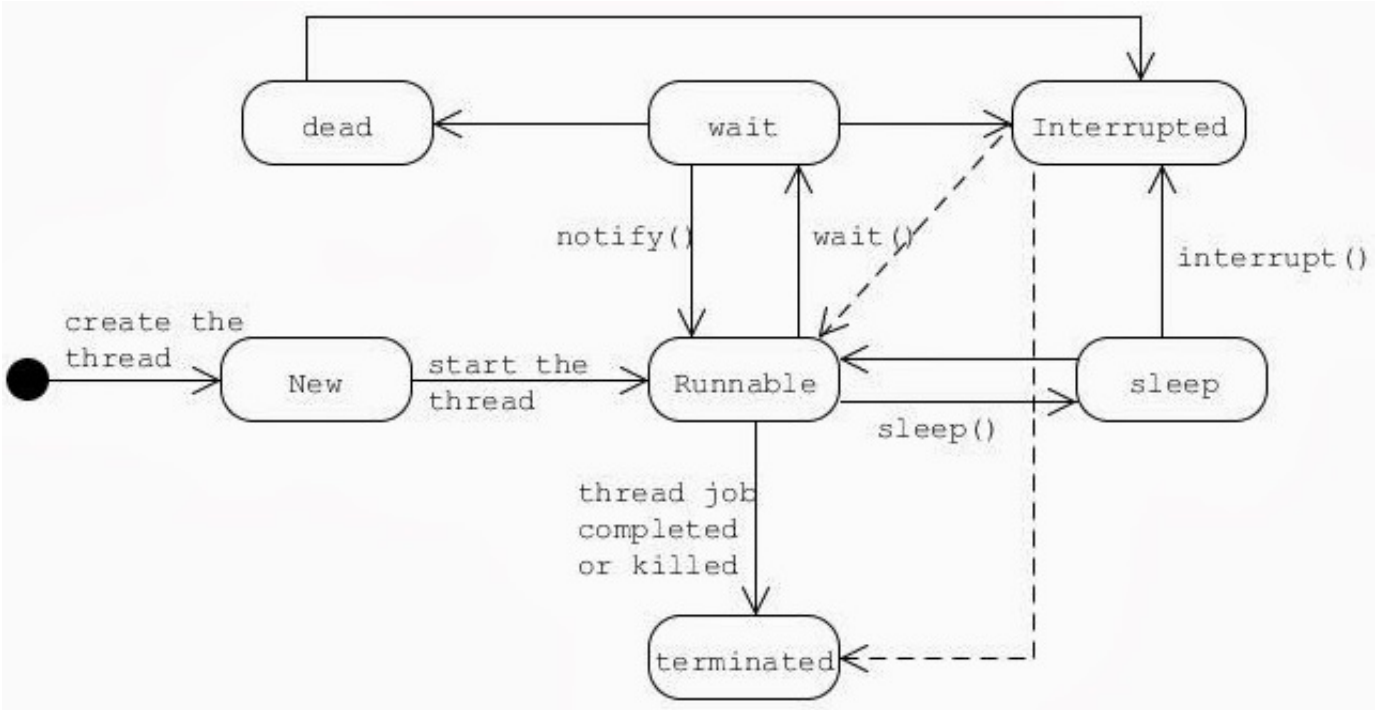
  Each thread has a unique ID (int value) which is provided by either the runtime or the OS; this is very helpful when you want to get the actual reference by.

- **Thread name**

  A name for the thread, it could be the same through many threads, also could be empty. It's useful to categorize threads belonging to a specific module or task.

- **Thread State**

  The current state of the thread. The state of a thread is determined by the State enumeration, and a thread state could be one of the following:



  - **Runnable**: indicates that the thread has started and is running its task.
  - **New**: indicates the thread has been created, but hasn't invoked (started) yet; when you just create an instance of the thread and haven't attempted invoke the `start()`.
  - **Blocked**: when a thread is in running state and is blocked because it is waiting for **locking** a resource which has been locked by another thread.

  - **Waiting**: when the thread is idle and is **waiting** for a signal (notify).
  - **Timed_Waiting**: when the thread is **sleeping** for an amount of time or is **waiting** for a signal from any another thread with **timeout**.
  - **Terminated**: when a thread either has finished its job, note that a **terminated** thread would not **start over from the beginning**, and it's ready for finalizing.
  - **Interrupted**: this indicates if an **interrupt** signal has been sent to the thread or not, note that this is not an **actual state**, a thread would get interrupted when it's in either running or waiting states (**not** in new and terminated states). You **cannot** get the interrupt state by the State enumeration.
  - **Dead**: when thread A is waiting for a signal from thread B, where thread B is waiting for thread A too, or maybe thread B has sent the signal just before thread A attempts to wait for it (this is the actual **dead lock**), here there is nothing left behind to signal the waiting thread. You need to just program your application in such a way that you ensure there is no dead state. Note that there is no property or method to determine if a thread is in dead state or not.

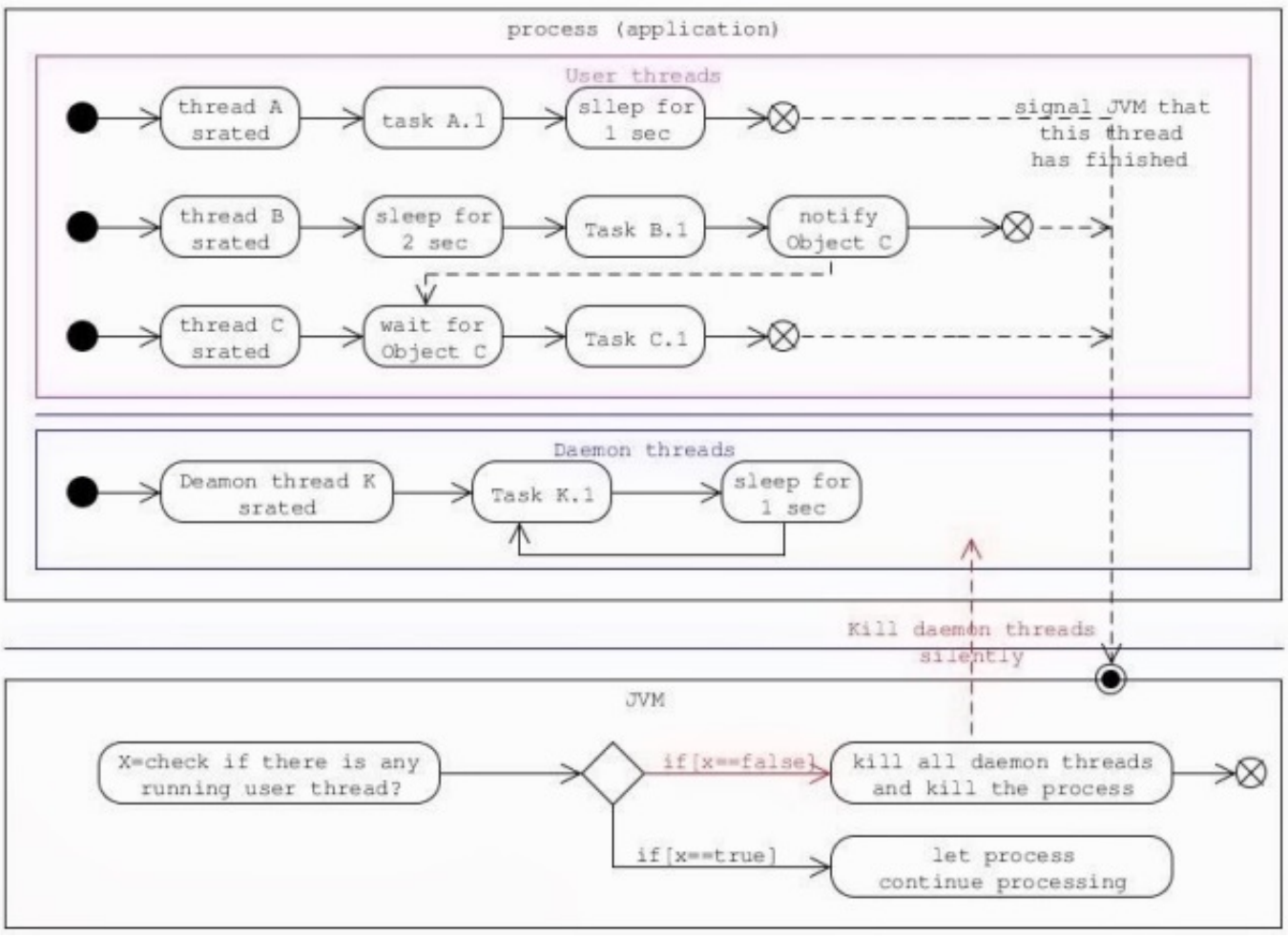
    Note: in some tutorials, they explain the dead as same as the terminate state, where this is okay, dead state is known as waiting state too, but according to my knowledge dead and terminated are two separated subjects

- **Thread Priority**

  It indicates the priority of the thread, a higher priority causes more process attention (context switch) to the thread, that could be one of the following:

  - `MAX_PRIORITY`
  - `MIN_PRIORITY`
  - `NORM_PRIORITY` (default value)

- **Daemon mode**: this one is used for setting that a thread should run in Daemon (background) mode or not. When a thread is in daemon mode, it means it runs until any other user thread (non-daemon) is alive in the application. Whenever there is no user-thread alive, then JVM kills the daemon threads by force (silently). You cannot even get any exception about the exit signal. These kinds of threads are useful when your application needs to have a background-service thread or an event-handler. These threads are usually run in **lower priority**. Once again, these threads are running within your process, but JVM **does not** look at them as user threads for checking the life of the process (check the diagram).



As the above diagram says, whenever a user thread gets finished, JVM tests if there is any user thread running yet. If there is not, then it kills every running daemon thread, so beware, **don't** try to change something in a daemon thread if you know it could get killed right now, it's better to signal daemons before every user thread gets finished. We will focus on daemon threads (examples) later, just keep reading.

Example

The following example is showing how to set and get thread properties, it's really simple.

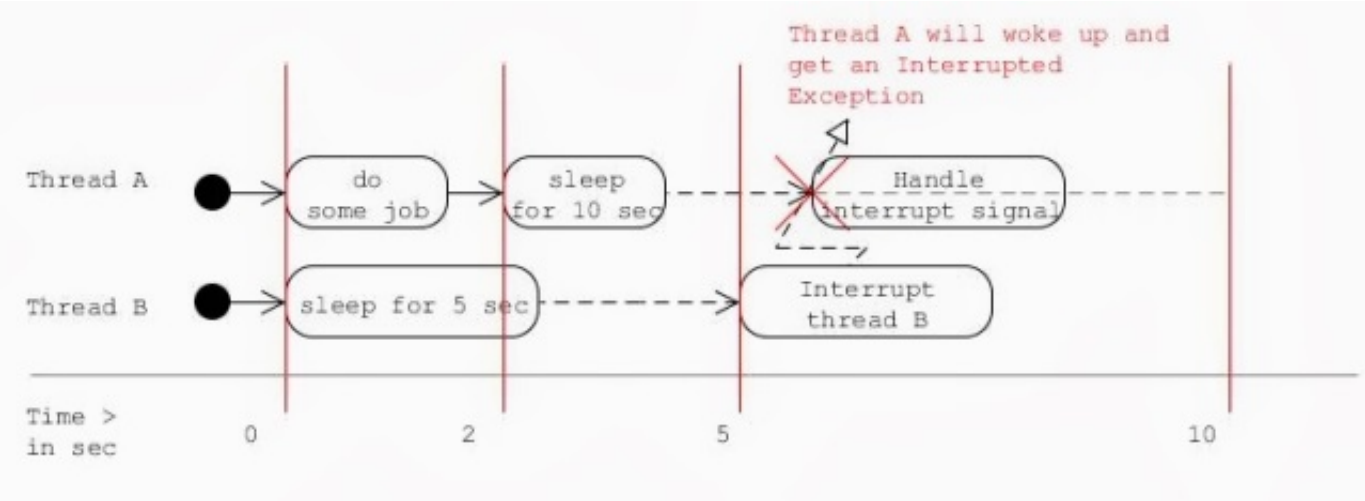
```
package arash.blogger.example.thread;
/**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
 public static void main(String[] args) throws InterruptedException {
  Runnable r0;//pointers to a thread method
  r0=new MyThread();
  Thread t0;
  t0=new Thread(r0);
  System.out.print("T0 thread state before thread start is "+t0.getState()+"\n");
  t0.setName("my lucky thread");
  t0.setPriority(Thread.MAX_PRIORITY);
  t0.start();
  Thread.sleep(1000);// wait for 1 sec here
```

```
  System.out.print("T0 thread state is "+t0.getState()+" right away\n");
  //t0.interrupt(); // this method will throw a Interrupt exception at the target thread.
  Thread.sleep(5000);
  System.out.print("T0 thread state is "+t0.getState()+" right away\n");
 }
}
class MyThread implements Runnable{
 private String arg;
 @Override
 public void run() {
  Thread t=Thread.currentThread();//returns the running thread, this thread.
  System.out.print("HI, my thread name is \""+t.getName()+
    "\" and my priority is \""+t.getPriority()+
    "\" and my ID is \""+t.getId()+"\"\n");
 try{
 Thread.sleep(5000);// wait for 5 seconds here
 System.out.print("Hallelujah, thread job has finished successfully!\n");
 }catch(InterruptedException e){
  /*if there is an interrupt signal
   * it defines by InterruptedException
   * so this catch determine whenever this thread got interrupt ex.
   */
  System.out.print("Oops, I've got interrupt message!\n");
  }
 }
}
```

Thread's Behaviors

Well, besides properties, a thread has some specific methods too, some of them belong to the thread class, some of them belong to a thread instance, and keywords. Well, the basic behavior (methods) of threads are listed below.

- **Start** (simultaneously): when the thread needs to run simultaneously with another thread(s), this operation is done by the thread `start()` method.
- **Sleep**: whenever a thread needs to sleep for a particular of time `Thread.sleep(time in ms)` would help, note that there is no warranty about the exact time, it could get more, the sleep process (reminded time) could get **skipped** if another thread **Interrupt**s the sleeping thread.
- **Interrupt:** it's used when a thread needs to interrupt another thread where it is in either wait (and `join()`), sleep, or even runnable states. For example, thread A is waiting for a signal, then thread B interrupts thread A, here the interrupt message:
    1. wakes up the target thread if it's either in wait (wait/join) or sleep state
    2. throws `InterruptedException` to the target thread to notify it about the interrupt signal
    3. sets the interrupted flag to true **if and only if** the thread was in running state.



and the code:

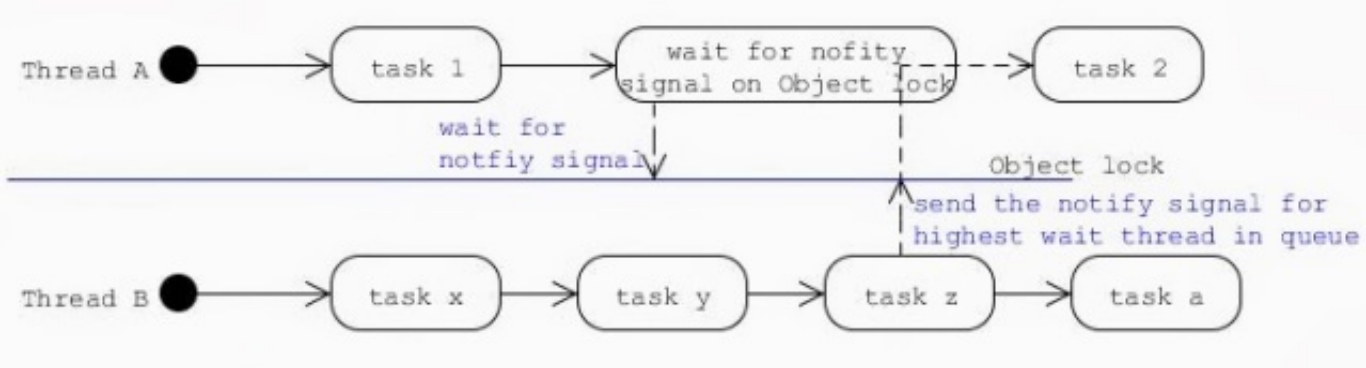
```
package arash.blogger.example.thread;
/**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
 public static void main(String[] args) {
  Runnable r0,r1;
  r0=new ThreadA();
  Thread threadA,threadB;
  threadA=new Thread(r0);
  r1=new ThreadB(threadA);//pass the thread A ref to thread B for interruption
  threadB=new Thread(r1);
  threadA.start();
  threadB.start();
 }
}
class ThreadA implements Runnable{
 @Override
 public void run() {
  System.out.print("A:I'm doing some operations\n");
  //.....some work
  try{// try to sleep
   System.out.print("A:let's sleep for 10 sec\n");
   Thread.sleep(10000); // sleep for 10 sec
  }catch(InterruptedException e)//while in sleep mode, interruption would happened
  {//handle if interrupted, do alternative tasks
   System.out.print("A:Oops, someone sent the interruption siganl and I will finish
myself.");
   return;//terminate the thread
  }
  System.out.print("A:I have to be a lucky guy, no any interruption message! thanks \n");
 }
}
class ThreadB implements Runnable{
 public ThreadB(Thread t){this.threadAref=t;}
 Thread threadAref;
 @Override
 public void run(){
  try {
   Thread.sleep(5000);//sleeps for 5 sec
   threadAref.interrupt();//comment it and see the differences
```

```
  } catch (InterruptedException e) {
   System.out.print("B: I've got an interruption signal, but I will continue my
task!\n");
  }
  System.out.print("B: nothing interesting here!\n");
 }
}
```

As the above figure says, both threads A and B start their tasks at the same time. Thread A tries to do something that would take 2 secs and sleeps for 10 secs, simultaneously thread B sleeps for 3 secs then wakes up and **Interrupt**s thread A. Note that when a thread is in sleep, join, and wait states:

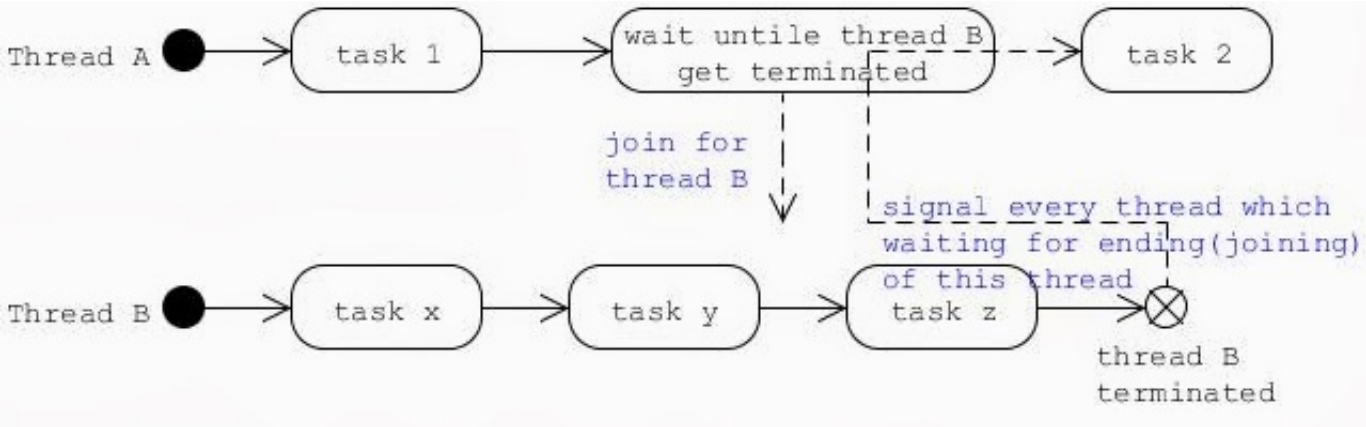
- every sleep, join, or wait operation should be surrounded with a try-catch block because of any interruption signal (exception)
- the interruption is just like a signal, it doesn't force the thread to shutdown

- **Getting the current thread**: you would access the current thread by `Thread.currentThread()`. It returns a reference to the currently executing thread object, but outside the thread you may need to check all the threads or have a references of the thread.
- **Wait, Notify, Notify All**: Well, I can say that the **Most Important** subject regarding threads is **Synchronization** between them, this is done by **Wait** and **Notify**. We will discuss about it in detail, but if I want to say it with a simple example, I would say whenever two or n threads need to synchronize with each other, a shared object needs to belong them, and now suppose that **thread A** has done operation 1 and has to wait for **thread B** to complete operation 2, here thread A after operation 1 **waits** for a notify signal on the **Object lock** and just after Thread B completes operation B, it sends the **notify** pulse with **Object lock** to thread A. It's really hard to explain by text, so check the following figure.



Note: It is possible an Object has more than one thread waiting on it. When three threads are waiting on an object, they stand on notify **queue** (not exactly a queue, dependents on priority too), it means for the next **notify** signal, the very first thread of the queue is released (signaled), but the **notifyAll** releases all threads in the queue. We will focus on this section, just keep reading.

- **Join**: it means as it says, when thread A joins thread B, it means thread A has to **wait** until thread B is finished (termination).




```
package arash.blogger.example.thread;
/**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
 public static void main(String[] args) {
  Runnable r0,r1;
  r0=new ThreadB();
  Thread threadA,threadB;
  threadA=new Thread(r0);
  r1=new ThreadA(threadA);//pass the thread A ref to thread B for interruption
  threadB=new Thread(r1);
  threadA.start();
  threadB.start();
 }
}
class ThreadA implements Runnable{
 public ThreadA(Thread t){this.threadBRef=t;}
 Thread threadBRef;
 @Override
 public void run(){
  try{
   System.out.print("A: Waiting for thread B get finished\n");
   threadBRef.join();//wait until threadB is alive
   System.out.print("A: thread B has completed almost");
  } catch (InterruptedException e) {//like the wait and sleep, join could get the
interrupt signal too
   System.out.print("A: interrupt signal has received!\n");
  }
 }
}
class ThreadB implements Runnable{
 @Override
 public void run() {
  try{
   Thread.sleep(10);//wait a bit, let the ThreadA goes first (not recommended)
   System.out.print("B:doing some work");
   for(int i=1;i<10;i++){
    System.out.print('.');
    Thread.sleep(1500);
   }
   System.out.print("\n B: Operation completed\n");
   Thread.sleep(2500);
  }catch(InterruptedException e)
  {
   System.out.print("B:interrupt signal has received!\n");
```

```
  }

 }
}
```

- **Yield**: I suggest you don't use it, use another way instead, and what does it mean? Very simple, if a thread calls `yield` it means the **OS can switch to another thread but the first thread still needs to work**. Here the host has just got the time for switching to another thread (if any, and if necessary, usually in the same priority). It's not the same as sleeping for 0 sec exactly, as I said, try to use another way instead. For example, there are two running threads with same tasks and priority and start time, and you want to try to finish them together (at least with a small time difference). At some time thread A realizes it has done 10% of its work while thread B has just done 4%. Here you cannot sleep for a small time, even 10ms, because it would cause that thread B does 30%, but you could tell the host to ignore this thread, and check another thread to switch to, but I still need to continue my work, switch back.

  I'm really sorry for the following example because it is a little hard for me to find a perfect example for yielding, but when you run the following for several times, you will face difference responses (because yield doesn't have a certain behavior), just comment the yield task and see the differences.

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
 public static ThreadA threadA;
 public static ThreadB threadB;
 public static void main(String[] args)
    throws FileNotFoundException, InterruptedException {
  System.setOut(new PrintStream("./out.res"));
  Runnable r0=null,r1=null;
  Thread a,b;
  r0=new ThreadA();
  r1=new ThreadB();
  threadA=(ThreadA)r0;
  threadB=(ThreadB)r1;
  a=new Thread(r0);
  b=new Thread(r1);
  a.start();
  b.start();
  a.join();
  b.join();
  System.out.flush();
 }
}
class ThreadA implements Runnable{
//imagine that two thread A and B cannot sync with other implicitly
 public int progress=0,i=0;
 @Override
 public void run() {
  System.out.print("Hello from 1st. thread\r\n");
  for(;i<10000;i++){//this is possible this thread perform even 2000+ loop in a
switch(turn)
   if(Core.threadB.progress + 5 < this.progress && Core.threadB.progress!=100){
    System.out.print("A: thread B is out of the date, try to switch [B(thread, loop):"+
       Core.threadB.progress+" , "+
       Core.threadB.i+"  A(thread, loop):"+
       Core.threadA.progress+" ,
       "+Core.threadA.i+"]\r\n");
```

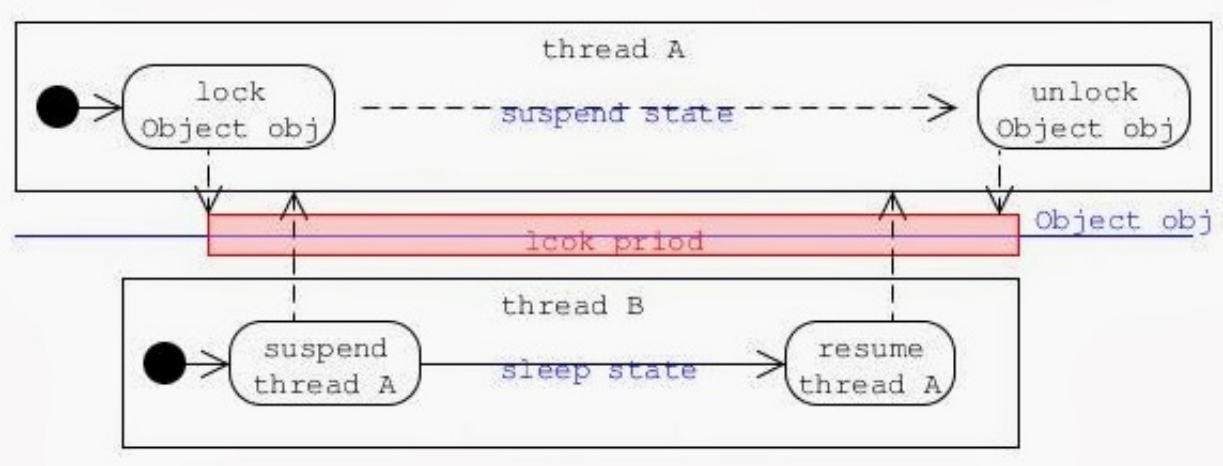
```
    Thread.yield();//tell host, ignore me now(a bit) and check the others
   }
   if(i%100==0){System.out.print("Thread A, progress :"+(i/100)+"\r\n");progress++;}
  }
 }
}
class ThreadB implements Runnable{
 public int progress=0,i=0;
 @Override
 public void run() {
  System.out.print("Hello from 2nd. thread\r\n");
  for(;i<10000;i++){
   if(Core.threadA.progress + 5 < this.progress && Core.threadA.progress!=100){
    System.out.print("B: thread A is out of the date, try to switch [B(thread, loop):"+
    Core.threadB.progress+" , "+
   Core.threadB.i+"  A(thread, loop):"+Core.threadA.progress+" , "+
     Core.threadA.i+"]\r\n");
    Thread.yield();//tell host, ignore me now(a bit) and check the others
   }
   if(i%100==0){System.out.print("Thread B, progress :"+(i/100)+"\r\n");progress++;}
  }
 }
}
```

The above code is not a cool example, but just check the result in the output file and imagine what is going on. Remember that if there are two threads that **cannot** sync with one another, using the `yield()` method may not work, you have to use another pattern and method instead, we will talk about it later.

Other Threads' Behaviors

We have mentioned about the most used (new-fashion) thread methods and behaviors, but there are still three deprecated (old-fashioned, but still work) methods here that you need to know the differences with the new ones, and why they have been deprecated.

- **Suspend and Resume**: the old fashioned options for wait and notify (synchronization), but there is a very big difference. First that Suspend pauses the thread (as well as Wait does) but **does not unlock** any synchronized (locked) resource, and this is dangerous, because for resuming the thread you need to have the exact thread references, while in wait and notify, you need locked object references. Also when a thread locks (synchronized) an object (A), and goes to wait for it, it means release it until the notify signal. After notify, it acquires the lock again.



- **Stop**: it looks like interrupt with a big difference. When you stop a thread, it means you are throwing an error (ThreadDeath[not exception] as a signal, same as interrupt) to the target thread, and **releasing all the locked resources**, like the interrupt you would catch the `ThreadDeath` error in a `catch` block and do the alternatives. This is dangerous because it is possible a thread gets stopped while it is locking a resource and is performing an update on it, so here the object is an inconsistent state.

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
  public final static Object o = new Object();
  public final static Object o2 = new Object();

  public static void main(String[] args) throws InterruptedException {
    MyThread t = new MyThread();
    Thread y = new Thread(t);
    y.start();
    Thread.sleep(2200);
    y.stop();// throws the ThreadDeath Error to the target thread
    synchronized (Core.o2) {
      System.out.print("Main thread: Could acquire the Object o2 lock.\n");
    }
  }
}

class MyThread implements Runnable {
  @Override
  public void run() {
    {
      System.out.print("Hi from MyThread!\n");
      synchronized (Core.o2) {
      // lock the o2 object, so the main thread should not access it
        while (true) {
          try {
            synchronized (Core.o) {// lock the o object
              for (int i = 0; i < 10; i++) {
                Thread.sleep(500);
                System.out.print(i + "\n");
              }
            }
          } catch (Exception e) {
            System.out.print("I've got an exception, but still holding o2 lock\n");
          }
        }
      }
    }
  }
}
```

In the above code, in `Mythread`, object `o2` has been locked by an infinitive loop. This means no thread would access it forever (like the main thread), but the `stop()` method causes throwing a `ThreadDeath` error to the target thread which here we aren't catching it, so this error releases all of the locked objects (`o`, `o2`) by the `MyThread` thread, but it is possible to catch `ThreadDeath` with a `catch` block, but note that, if you catch the error, then the locked object won't get released because your thread still has the control, like shown below.

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {

  public final static Object o = new Object();
  public final static Object o2 = new Object();
  // Source code level: 7
  public static void main(String[] args) throws InterruptedException {
    MyThread t = new MyThread();
    Thread y = new Thread(t);
    y.start();
    Thread.sleep(2200);
    y.stop();// throws the ThreadDeath Error to the target thread
    synchronized (Core.o2) {
      System.out.print("Main thread: Could acquire the Object o2 lock.\n");
    }
  }
}

class MyThread implements Runnable {
  @Override
  public void run() {
    {
      System.out.print("Hi from MyThread!\n");
      synchronized (Core.o2) {// lock the o2 object, so the main thread should
                              // not access it
        while (true) {
          try {
            synchronized (Core.o) {// lock the o object
              for (int i = 0; i < 10; i++) {
                Thread.sleep(500);
                System.out.print(i + "\n");
              }
            }
          } catch (Exception | ThreadDeath e) {// ThreadDeath for handling any stop
signal.
            System.out.print("I've got an exception or Error, but still holding o2
lock\n");
          }
        }
      }
    }
  }
}
```

The difference between this one and the previous is that, the second one also catches `ThreadDeath`, so the thread still has control and the `o2` object will be in lock state, and the next `while` loop gets started.

Once again, try not to use these deprecated methods, however you know the differences and the behaviors if you are planning to use them, so just keep your eye on any unexpected error and signal it would thrown.

Synchronizing Threads

Well, I would say this is the most important section in this tutorial, synchronizing two thread is really important when you are developing a multi-threaded program. For synchronizing two threads you need a shared **Object** (no primitives) and **cannot be null**. The lock object is usually an actual `Object` class and is called the **lock** Object. This object(s) should be visible (shared) to threads that want to sync with each other, thread A **waits** (in queue) on the **lock** object for the **notify** signal from thread B that causes thread A to release from the lock and continue its work.

Notes!!!

- More than one thread would wait on a lock object, and the next notify just releases **one** thread from the waiting queue (dependents on priority, JVM chooses)
- **notifyAll** causes **all of the waiting threads to get released from the queue**
- When a thread is in wait state, it means it's blocked, and someone has to send the notify signal to the thread in order to release the thread for the rest of the work, so beware of the **Dead**(dead-lock, infinity wait) situation

How to Synchronize Two Threads With Each Other?

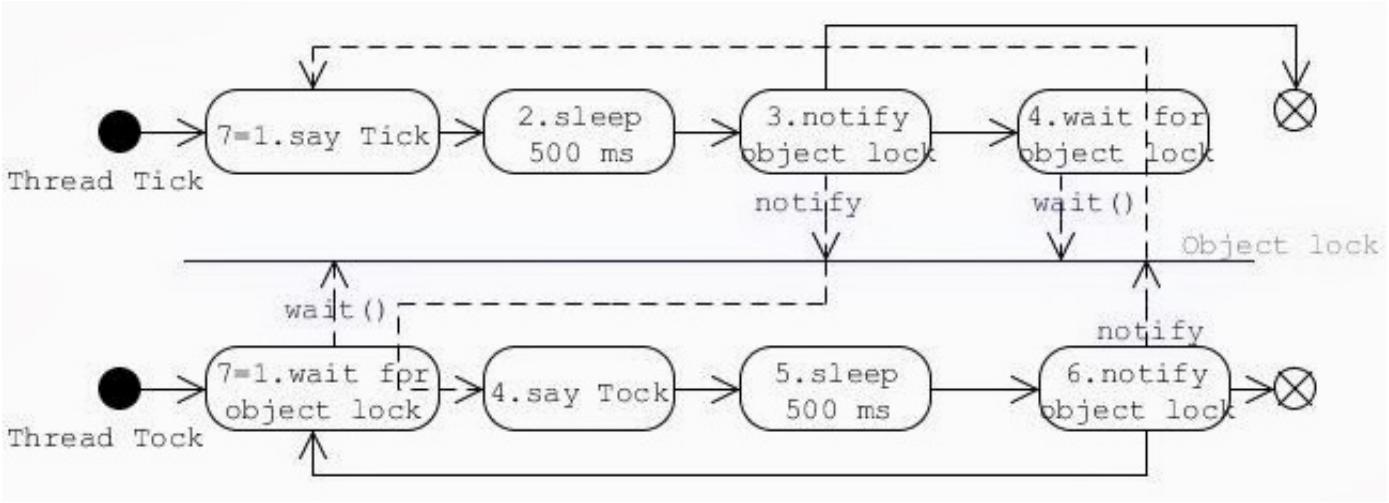
Okay, now it's kicking time, let me explain it with a simple example and the process of the program should flow.

Example

A **Tick Tock** application. Okay now let's look closer at this application, where is the start point? How many threads? Does it need synchronization? etc... Well, a Tick Tock application starts with **saying Tick**, then 0.5 sec waiting, then **saying Tock**, next 0.5 sec waiting, and start over, Tick, Tock, Tick, Tock,....., and now let's identify the processes:

1. Creating an Object lock for synchronization
2. Creating two threads Tick and Tock and pass the lock object to them
3. Start the threads together
4. (In a for loop) Tick says Tick, and Tock waits for thread Tick for notify signal on lock object
5. Tick sleeps for 0.5 seconds, while Tock is still waiting for Tick signal
6. Tick sends the notify signal to object lock, and Tock gets released from the lock
7. Now Tock says Tock, and Tick waits for thread Tock for notify signal on lock object
8. Tock sleeps for 0.5 seconds, while Tick is still waiting for Tock signal
9. Tock sends the notify signal to object lock, and Tick gets released from the lock
10. Back to step 4! Until for loops get finished.

and the diagram would be:



and now before we go for the code, we need to know about **locking** in Java, and it would be easy, so just keep reading.

Synchronized Block

In Java sometimes you need to lock a resource, because it needs to ensure that no thread reads or writes on that object, so there is a locking operation required by the related thread. It's really easy, it goes like this with a synchronized block:

```
synchronized(lock_Object){
//here no any thread would access (read, write) the lock_Object except the current thread.
}
```

For waiting and notify on a lock object, it should be locked (synchronized) before either wait, notify, or notifyAll. As shown below:

```
//Object will be locked by this thread if it's free to lock, else has to wait until released!
synchronized(lock_Object){
lock_Object.wait();//release the lock, and acquire it again after get notified
}
//lock_Object2.wait();// Error!, should get synchronized like above for either waiting or
notifying
```

Important note: As we said, when a thread locks (synchronizes) a resource, no thread would access that resource. If a thread invokes the `wait()` method of an object (lock), that object is released (release the lock) of the object in order to enable another thread to acquire lock on the object and notify the waiting thread. This situation is **not necessary** for join, sleep, and yield methods, just belongs to wait. If in a synchronized block, a thread sleeps for 10 years, it means the synchronized object is locked for 10 years. Okay we have got everything to know to impl our example now, and what are we waiting for, check the following.

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
 public static Object lock=new Object();
 public static void main(String[] args) throws FileNotFoundException, InterruptedException {
```

```
  Runnable r0=null,r1=null;
  Thread a,b;
  r0=new Tick();
  r1=new Tock();
  a=new Thread(r0);
  b=new Thread(r1);
  a.start();
  b.start();
  a.join();
  b.join();
  System.out.print("Hallelujah, threads finished!");
 }
}
class Tick implements Runnable{
 @Override
 public void run() {
  try{for(int i=0;i<20;i++){
   System.out.print("Tick ");
   Thread.sleep(500);
   synchronized (Core.lock) {
    Core.lock.notify();//notify tock, it's his turn  [1]
    Core.lock.wait();//and wait until tock notify you for next cycle  [4]
   }
    }
  }
  catch(InterruptedException e){}
 }
}
class Tock implements Runnable{
 @Override
 public void run() {
  try{
   for(int i=0;i<20;i++){
  synchronized (Core.lock) {
    Core.lock.wait();//wait for tick completes its turn  [2]
    }
    System.out.print("Tock ("+(i+1)+")\n");
    Thread.sleep(500);
    synchronized (Core.lock) {
     Core.lock.notify();//my turn completed, signal tick for next cycle[3]
    }
     }
   }catch(InterruptedException e){}
 }
}
```

Any questions? Always beware that wait, notify, and notifyAll need to be invoked inside a synchronized block where they are locked by the synchronized block.

Synchronized Methods

A method is called synchronized when that method wants to **locks its object (this)**. It looks like a method that locks (synchronizes) its object (this), so other threads should wait until the method gets released. Please note that while a synchronized method is running, because it has locked the host object, no thread would invoke **the members belonging** to the host object (this), so if a synchronized thread needs to wait, it should wait for the host object (this), and here wait and notify don't need to be surrounded by a synchronized block, because a synchronized method does it implicitly (if and only if it's about wait and notify the host object).

```
public synchronized void method(Object arg){
//it means this object will be locked by this block (method)
}
public void method(Object arg){
  synchronized(this){
  //this is same as above, but it's not recommended
  }
}
```

Let's have a simple example.

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
  public static Synch s = new Synch();
  public static void main(String[] args) throws InterruptedException {
    new Thread(new MyThread()).start();
    Thread.sleep(1);// ensure that MyThread go first
    new Thread(new MyThread2()).start();
  }
}

class MyThread implements Runnable {
  @Override
  public void run() {
    try {
      Core.s.intoduceFriends();// calling a synchronized method
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}

class MyThread2 implements Runnable {
  @Override
  public void run() {
    Core.s.whoAmI();// I should wait for 25 sec
    // whole of the Synch object is locked now!
  }
}

class Synch {
  public synchronized void intoduceFriends() throws InterruptedException {
    System.out.print("Fiends are {Danial, Pedram, Armin, Aria, Manochehr, Farzad, Aidin, and
you!}\n");
    Thread.sleep(25000);
    // sleeping 25 sec in a synchronized methood means
    // 25 sec of locking the host object (this), no any member will be
    // accessible!
  }
  public synchronized void whoAmI() {// surprise
    System.out.print("This example has prepared by Arash M. Dehghani\n");
  }
}
```

Example

Let's have the above example (tick-tock), but now synchronize threads by synchronized methods.

```
package arash.blogger.example.thread;
/**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
  public static Lock lock=new Lock();
  public static void main(String[] args) throws FileNotFoundException, InterruptedException {
   Runnable r0=null,r1=null;
   Thread a,b;
   r0=new Tick();
   r1=new Tock();
   a=new Thread(r0);
   b=new Thread(r1);
   a.start();
   Thread.sleep(1);//ensure that tick is going first!
   b.start();
   a.join();
   b.join();
   System.out.print("Hallelujah, threads finished!");
  }
 }
 class Tick implements Runnable{
  @Override
  public void run() {
   try{for(int i=0;i<20;i++){
    System.out.print("Tick ");
     Core.lock.tickCycle();//notify tock, it's his turn  [1]
     }
   }
   catch(InterruptedException e){e.printStackTrace();}
  }
 }
 class Tock implements Runnable{
  @Override
  public void run() {
   try{
    for(int i=0;i<20;i++){
     Core.lock.tockCycle();//wait for tick completes its turn  [2]
     System.out.print("Tock ("+(i+1)+")\n");
      }
    }catch(InterruptedException e){e.printStackTrace();}
  }
 }
 class Lock{
  //synchronized method automatically lock and release this object
  public synchronized void tickCycle() throws InterruptedException{
   Thread.sleep(500);
  }//there is no need for signal notify or wait for it here.

  public synchronized void tockCycle() throws InterruptedException{
   Thread.sleep(500);
  }
}
```

As in the above example (I recommend the previous one), we have a new member Class Lock here that contains two synchronized methods. In the `tickCycle()` method, it sleeps for 500 ms, and because it's a synchronized method, no thread would invoke any member of this class, so the tock thread has to wait until

the `tockCycle()` member gets released (lock object in fact).

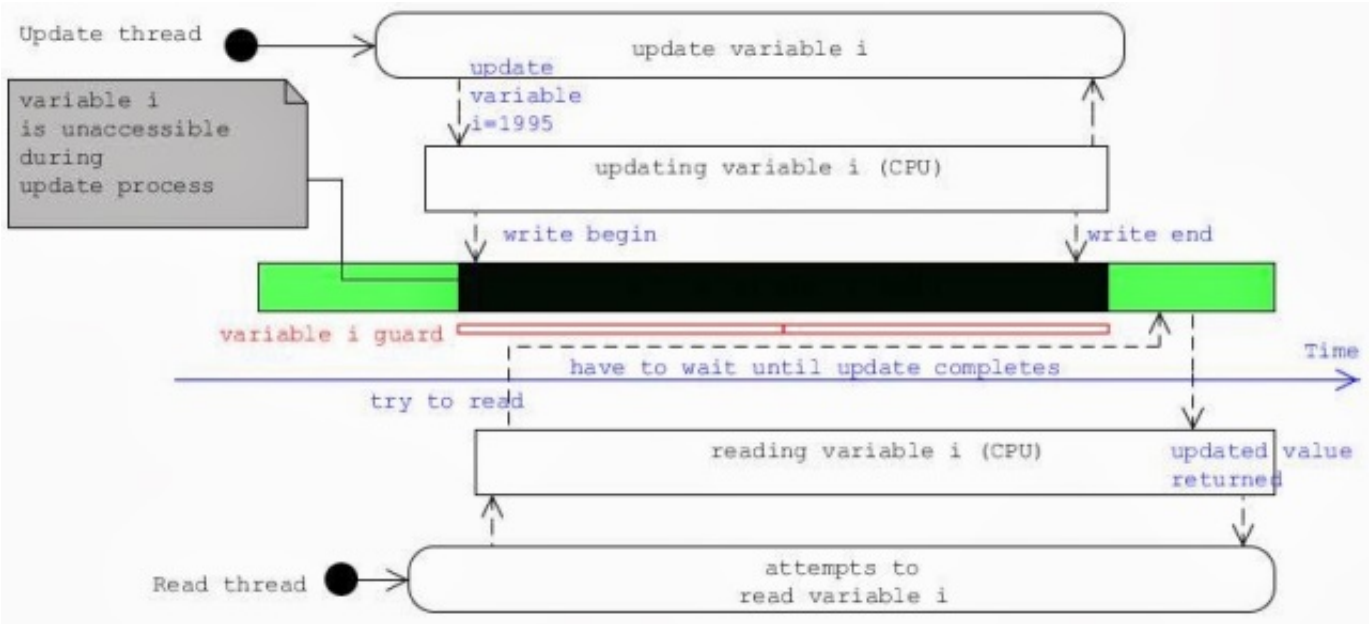
Exercises

And now, it's your turn, try to impl the following case studies, and feel free to contact me for any issues or even verifying your impl.

- Just try to extend the above example with Tick Tock Tack!
- A program that starts to write 10 large file in the hard disk with a heavy processing of data (as you wish, could be combination of shifting, square, sqrt, etc...), and here a thread should display the overall progress of the process, note that the process resolution is 10%.
- A graphical program (Swing, AWT, ...) that at the top of the screen is a moving circle, and you have to hit it with your gun (cursor). The user has to get 20 scores before time runs out (2 min maybe!).

volatile Keyword

This keyword (feature) was added by JSE 5+. They added it for multi-threaded applications, and what does it do? In a multi-threaded system, it is possible that two threads want to set and **get** a value of a variable at a time, so here inconsistency would appear. For example, two threads, one of them is updating a value `i`, and another one wants to read the value `i`, it is possible that the second thread reads the old value, so there is no warranty that forces the second thread to read (wait for) the new value, except get it via a method and lock it by the synchronized keyword (single thread approach). Here if a field is volatile, it means the reading or setting (no read-modify set) threads ensure that they are reading the very last value of the variable. This is not recommended for read-modify (like i++, i=i>>2, i*=2) changes, **because volatile does not lock the object**. volatile is really helpful when there are some threads updating a value, and simultaneously a thread needs to read the value (like an auction system). volatile ensures that the reading thread will get the very last update, it waits the thread until another update thread gets finished if any.



And now when should we use volatile? And when should use synchronized block instead?

- If threads need to **just read** a value, and you want to ensure them the latest updated value, use volatile.

- If threads attempt to read and update a value without the need to know the current value (like i=10, i='A', i=1.990E4D), use volatile.
- If threads attempt to read and update a value and the update depends on the current value (like i=i+1, i%=7, i=1.994E4D-i*2), use a synchronized thread

It is possible to replace all of the volatile ways to synchronized way, but synchronized places a lock (block) and it would take time, volatile is faster than synchronized, **but remember it does not block the variable**. A volatile variable would be either primitive or objective with null values too, while a synchronized block just works with non-null (for synchronizing) objectives.

How to Set a Variable to Volatile?

This is really easy, just set as volatile with the variable's declaration, like below.

```
public class Invoker{
public volatile Object obj;//it could be null
private volatile int id;//it could be primitive too
public int getId(){
   return this.id;//because id is volatile, so if there is an update in progress,
   // it waits for the last update, then return the updated value.
   }
}
```

Example

Now let's have some examples, decide either volatile or synchronized approach would help for each one.

Last User Logged on

This is an application that would track (set and get) the last user ID logged into the system, impl the desired system. And now we have to decide if this would be multi-threaded? Yes of course, because the data (last user ID) would get updated and read from different systems and requests concurrently. Okay, first as is mentioned, the user ID gets updated to a new value, and it **doesn't** depend on the old value, so here the volatile approach would help with setting and getting the value, so the code would be like this.

```
public class SysMgr{
  private volatile String lastUserId=null;
  public void set(String lui){ this.lastUserId=lui; }
  public String get(){ return this.lastUserId;}
}
```

In the above code, if thread(A) wants to get the lastUserId value, and simultaneously another thread(B) is updating it, then thread A will be waited (blocked) till thread B completes the update, then returns the new value to thread A, and all of these because of the volatile keyword.

Total Login Count

An application that would count, how many times the system login has.

Here this example is a bit same as the previous example, except, when you want to count, you want to update an **existing** value, in fact, the update process is **read-modify**, it depends on the current value, so here volatile would not help, because volatile does not lock the variable. But for reading it is okay, then the

code would be like this, variable definition will be volatile for readers and direct-set value, and a synchronized method to ensure there is only one thread updating the value.

```
public class Statistic{
  private volatile long logins=0L;
  public synchronized void newLogin(){logins++;}//because this method is synchronized, maximum
of 1 runnable is exist
  public long getValue(){return this.logins;}//the variable logins is volatile, so it ensures
the very last update value
  public void reset(){this.logins=0;}//it could be without locking, because variable is
volatile
/*
* some private members
*/
}
```

The `getValue()` is not synchronized, and this is not required, because for any reading, this ensures your committed (new) value will be returned. Note that the `getValue()` method should be synchronized if the login variable was not volatile.

Relative and Absolute Update

Here we are going for an application that would update a variable directly (modify only) or increment it by one (read-modify). Well this is going to be easy, we could separate these two tasks into two synchronized and non-synchronized methods.

```
public class Mgr{
  private volatile long l=0L;
  public synchronized void increment(){l++;}//this is read-modify update, so needs the lock
  public void set(long l){this.l=l;}//this is just modify update, so could be done with
volatile
}
```

Well, you have nt focus like those in your application, if this is not going to be a large project. You just have to know about the differences, when to use lock and when to use volatile, and when both!

Notes

Okay Okay Okay, I would say, now you can write any multi-threaded application by hand, but there are still some important things that you need to know about. Like patterns, issues, managements, and even how to make a coffee after this tutorial.

Main Thread

If your application has a main method, then JVM creates a thread (main thread), and runs your main method with this thread.

Creating Thread

In Java there are two ways for creating (declaring) threads, implementing the `Runnable` interface or extending the `Thread` class, and the first way is more common. Like shown below:

```
class MyThread implements Runnable{//recommended
 @Override
 public void run() {}
}
class MyThread2 extends Thread{
 @Override
 public void run() {}
}
```

Daemon Threads

As we mentioned, there are two kinds (by the running method) of threads in Java, user-threads and daemon-threads. As mentioned, JVM kills and purges the memory of your application when there is no user-thread running, and kills daemon-threads silently, so please beware about daemon-threads, do not change something if you know the application could be finished right now, because daemon threads would not receive any interrupted or thread death signal. Either do not change anything within these threads, or signal them implicitly before your user threads get terminated, just check the code below.

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
    public static void main(String[] args) throws InterruptedException {
        MyThread t = new MyThread();
        Thread y = new Thread(t);
        y.setDaemon(true);
        // you have to set it before you start it!, set it as false
        // and run the code again, see the differences

        y.start();
        Thread.sleep(5200);
        System.out.print("The very last result from our daemon-thread: "+t.res+"\n");
        System.out.print("Main thread has terminated, no any user thread available!\n");
    }
}

class MyThread implements Runnable {
    long res = 1L;

    @Override
    public void run() {
        {
            System.out.print("Hello, I am a daemon thread, wohaha!\n");
            while (true) {
                try {
                    for (int i = 0; i < 10; i++) {// if I be a lucky thread, I will reach the
10
                        Thread.sleep(1000);
                        res += i;
                        System.out.print("Res right a way: " + res + "\n");
                    }
                } catch (Exception | ThreadDeath e) {// catch for any stop() or interrupt()
signal
                    System.out.print("I will never get any signal by JVM :(\n");
                }
            }
        }
```

```
    }
}
```

As the above code says, `MyThread` is a daemon thread. You have to set a thread as daemon before you start it, `MyThread` has a `for` loop that takes 10 sec to finish, but because it is a daemon, it will killed by JVM when the main thread gets terminated.

Shutting Down a Thread

As we mentioned, there is no clear method for shutting down a thread, so you cannot force a thread to shutdown. So you have to design a way to determine that the Thread should stop working, one of them is using the `interrupt()` method and interrupted state of the thread, but this is not recommended, because interrupt has been designed for waking up a thread which is sleep, join, or wait state. Please note when you are invoking the interrupt method, it could have different behaviors.

- if thread is in wait, join, or sleep state, then the rest of the sleep/wait state is omitted and an `InterruptedException` sent to the thread by JVM. JVM **doesn't** set the interrupted flag to true.
- if thread is blocked by an IO operation (reading file, stream, ...), then the rest of the sleep/wait state is omitted, the IO channel gets closed, and a `ClosedByInterruptException` sent to the thread by JVM. JVM **doesn't** set the interrupted flag to true.
- if thread is in runnable state, then just set the interrupted flag to true (we can use it as a shutdown signal)
- if thread is in new state (haven't started yet), then nothing happens!

So here we could use the interrupted flag as shutdown flag to determine the thread has received the shutdown signal or not; for example, check the following:.

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {//Just run it for several times, you will faced with different results
    public static void main(String[] args) throws InterruptedException {
        MyThread t = new MyThread();
        Thread y = new Thread(t);
        y.start();
        Thread.sleep(2);// wait 2 ms, see how much your system strong is in 2 ms?
        y.interrupt();//set the interrupted flag to true
        System.out.print("Main thread: I've send the interrupt(shutdown) signal\n");
    }
}

class MyThread implements Runnable {
    int i=1;
    @Override
    public void run() {
        while(Thread.currentThread().isInterrupted()==false){//until shutdown signal haven't
sent
            double a=0.0D;
            //some big load!
            for(int k=100;k>i;k--){
                a=k*k*2/5*Math.pow(2, k);
                a=Math.sqrt(a);
            }
            System.out.print("Res("+i+"): "+a+"\n");
            i=i+1;
```

```
        }
        System.out.print("I've got an interrupt signal :(\n");
    }
}
```

Here in above code, as you see `MyThread` continues its work until another thread sets its interrupted flag to true, which means stop working. Note that in `MyThread` there is no try-catch, because as we mentioned, if a thread is in running state, its interrupted flag is set to true (no exception, not the same for the `stop()` method!).

While the above code works without any problems, it is recommended to use an implicit way to signal a thread to stop its working, that we would impl it like this:

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
    public static void main(String[] args) throws InterruptedException {
        MyThread t = new MyThread();
        Thread y = new Thread(t);
        y.start();
        Thread.sleep(2);// wait 2 ms, see how much your system is strong in 10 ms?
        t.shutdown();
        System.out.print("Main thread: I've send the interrupt(shutdown) signal\n");
    }
}

class MyThread implements Runnable {
    private volatile boolean shutdown=false;//use a user flag
    public void shutdown(){this.shutdown=true;}
    //a method instead of interrupt method for sending the shutdown signal.

    int i=1;
    @Override
    public void run() {
        while(this.shutdown==false){//until shutdown signal havn't sent
            double a=0.0D;
            //some big load!
            for(int k=100;k>i;k--){
                a=k*k*2/5*Math.pow(2, k);
                a=Math.sqrt(a);
            }
            System.out.print("Res("+i+"): "+a+"\n");
            i=i+1;
        }
        System.out.print("I've got an interrupt signal :(\n");
    }
}
```

The differences is just about the shutdown flag, one of them via interrupted flag, and another one with a customized user flag, and I recommend the second one.

wait(1000) VS. sleep(1000)

We have mentioned the `wait()` and `sleep()` methods, let's review the behaviors of the two.

wait()

It releases the synchronized (locked) object, and waits until another thread notifies (notify, stop, or interrupt) it, then after it gets notified (if by notify) tries to acquire the lock again and continues its work. Note that a waiting thread has to acquire its lock again on the related object.

sleep()

ID doesn't release any locked object, and sleeps for a specified amount of time, it could get canceled (wake up) by either stop or interrupt signals.

But here we could wait (`wait(max_of_wait_time)`) for a notify signal, but with a **maximum of time**. It means the thread releases the locked object and waits for the notify signal, but it will notify itself if no thread notifies it by timeout.

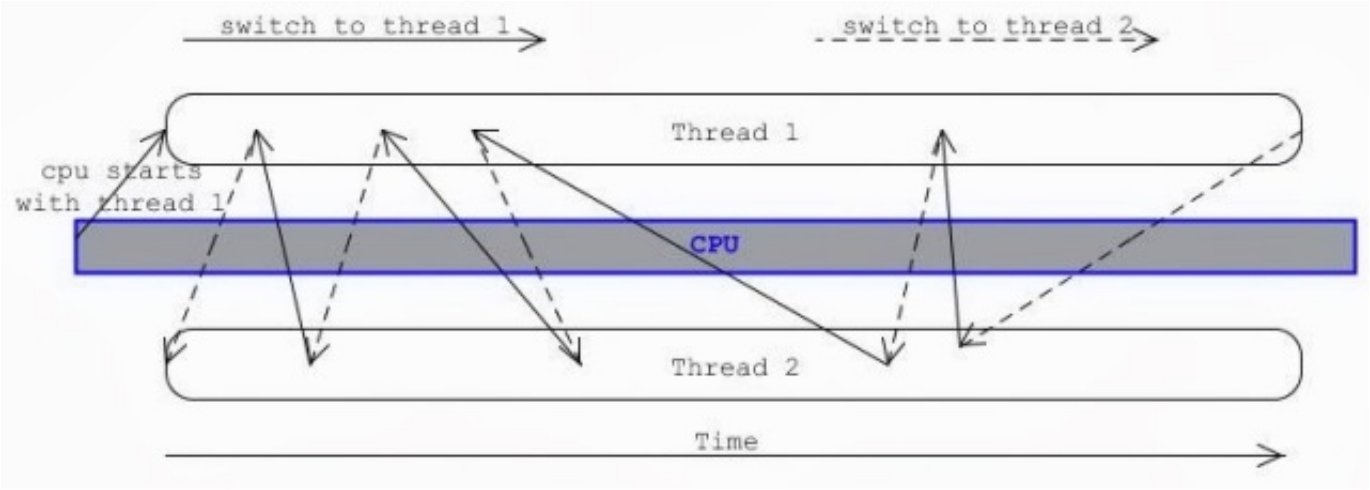
```
synchronized(lockObj){
   lockObj.wait();
   //releases the lockObj(just lockObj) and waits
   //for notify signal, even if it could be 2 years
}

synchronized(lockObj){
   lockObj.wait(20000);//releases the lockObj and waits for notify signal,
   // but it notify (and acquires lock again) itself after 20 sec
}

synchronized(lockObj){
   Thread.sleep(20000);//just sleep for 20 sec
}
```

So the only difference between `wait(1000)` and `sleep(1000)` is that `wait(1000)` sleeps for 1 sec and releases the locked object too, while `sleep()` doesn't release the lock. I've read in some tutorials that we should try to use `wait` instead of `sleep`, but I don't recommend it.

Switching Between Threads (untrustful yield())

What does switching mean exactly? It's a really easy peasy subject, a single thread/single core CPU would run only one job at a time, and it has to switch to another thread and do some tasks of other threads too, and so how do we see everything smooth? Because our CPU is really powerful (cheers), switching between threads would be even less than 10 nanoseconds!

When you want to switch between your threads, you would do it implicitly (direct) by `wait` and `notify`, or do it explicitly (in-direct). We use notify/wait approach when even one step of a thread is important for us, for example should run step B after step C. The following diagram is showing possible thread switching between two un-synchronized threads, but note that it's even possible CPU completes whole of the thread one then switches to thread two, or does 1% of thread 1, switches to thread 2 and does it 100%, then switchs back to thread 1 and completes the thread 1. well as you see there is no warranty that ensures you two (same time start) concurrent thread get finished at a time too, unless you synchronize them with each other.



But when there is no(no need to) any wait and notify method, how to switch(try switch) to another thread? we have mentioned about yield() method, but as we said do not trust yield() method, because this method calls underlying(OS) functions that means skip this one and switch to others. but sometimes you see no any differences. For example just check this example out.

```
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
  public static void main(String[] args) throws InterruptedException {
      new Thread(new MyThread()).start();
      new Thread(new MyThread2()).start();
      //no one knows when this print will get invoked! after thread 1? or even 2? or even after
them?
      System.out.print("Main Thread: I started the threads :)\n");//main thread sentence [S]
  }
}

class MyThread implements Runnable {
  @Override
  public void run() {
    for(int i=0;i<20;i++){
      System.out.print("Thread 1: "+i+"\n");
    }
  }
}

class MyThread2 implements Runnable {
  @Override
  public void run() {
    for(int i=0;i<20;i++){
      System.out.print("Thread 2: "+i+"\n");
    }
  }
}
```

Just run the above code many times and you will see many results, I even got this result (whole thread 1, whole thread 2, and then main thread).

Okay now let's have a use case, in the above code, we want to try to wait thread 1 and thread 2 for a bit, and let the main thread says its sentence ([S] section) first, then switch back to them, first let's have it with yield method, in the thread 1, and thread 2 we call yield method, it means skip this (thread 1 and thread 2),

and switch to another thread. check the new threads methods.

```
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
class MyThread implements Runnable {
  @Override
  public void run() {
    Thread.yield();//tells host skip me, switch to others.
    for(int i=0;i<20;i++){
      System.out.print("Thread 1: "+i+"\n");
    }
  }
}
```

if you ask me, I would say there is no any differences! and this is because many users try not to use it, but here what should we do? sometimes you are walking with your dog, you tell him stop me for a while(yield() method), and you see no differences!, no responses!, then you try to do it yourself, you stop yourself for a bit, then another thread would get switched, here you stop yourself for a very small of time (even less than 1 ms), so except calling yield() method (if doesn't work) why not sleeping for a bit!? so our new code would be like this:

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
  public static void main(String[] args) throws InterruptedException {
      new Thread(new MyThread()).start();
      new Thread(new MyThread2()).start();
      //no one knows when this print will get invoked! after thread 1? or even 2? or even
before them?
      System.out.print("Main Thread: I started the threads :)\n");// main thread sentence [S]
  }
}

class MyThread implements Runnable {
  @Override
  public void run() {
    try {
      Thread.sleep(0, 50);//if you don't sleep me, I sleep myself for 50 nano seconds! , wohaha
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    for(int i=0;i<20;i++){
      System.out.print("Thread 1: "+i+"\n");
    }
  }
}

class MyThread2 implements Runnable {
  @Override
  public void run() {
    try {
      Thread.sleep(0, 50);//but still there is no warranty!
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

```
    for(int i=0;i<20;i++){
      System.out.print("Thread 2: "+i+"\n");
    }
  }
}
```

In the above code, we just replaced the yield method with sleeping for 50 nano seconds! but still **I'm not sure** that main thread sentence [S] goes first, there is more chance that main thread goes first here, but I say again, there is still no any warranty that force thread1 and thread 2 go after thread main even if you sleep for 2 seconds instead of 50 nano seconds, unless you synchronize them.

note that it dependents on your CPU too, if your CPU is strong enough then 50 nano seconds is good enough to switch to another, but another CPU may needs more!

Advanced Topics

There is no room for low-performance, so let's talk professionally, here we want to talk about advanced topics, but remember this article for beginners.

Manage Your Memory

Each thread has a private memory which is accessible to itself(methods, classes, etc...), and a shared memory which is belong to every threads, note that these resources (memory) would get separated into two types, some of them are generated and managed by you (developer), and some of them (like thread id, stacks, properties, etc...) are belong to host (either JVM or OS), so beware of creating to much threads, it would eat whole of your memory very easy. executing a sequential(serial, single-thread) application is more more more easier than a multi-thread application for the CPU, while you are able to run your application by GPU (with restricted functionality). now just run the following code to see yourself how many threads would disturb your system, and watch your system memory and performance.

```
package arash.blogger.example.thread;
  /**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
 public final static Object o=new Object();
 public static void main(String[] args) throws InterruptedException {
  for(int i=0;i<5000;i++){//what if running 5000 threads, watch your system!
   new Thread(new MyThread()).start();
  }
 }
}
class MyThread implements Runnable{
 //there are no any user variable
 @Override
 public void run() {
  //some work
  try{
   synchronized (Core.o) {
    Core.o.wait();
   }
  }catch(Exception e){

  }
```

```
  }
}
```

What if there are 5000 of threads in memory!?, some are waiting for garbage collector, some are waiting, some are sleeping, and some are idle, it's just a disaster, but it is would get solved with patterns, just keep reading.

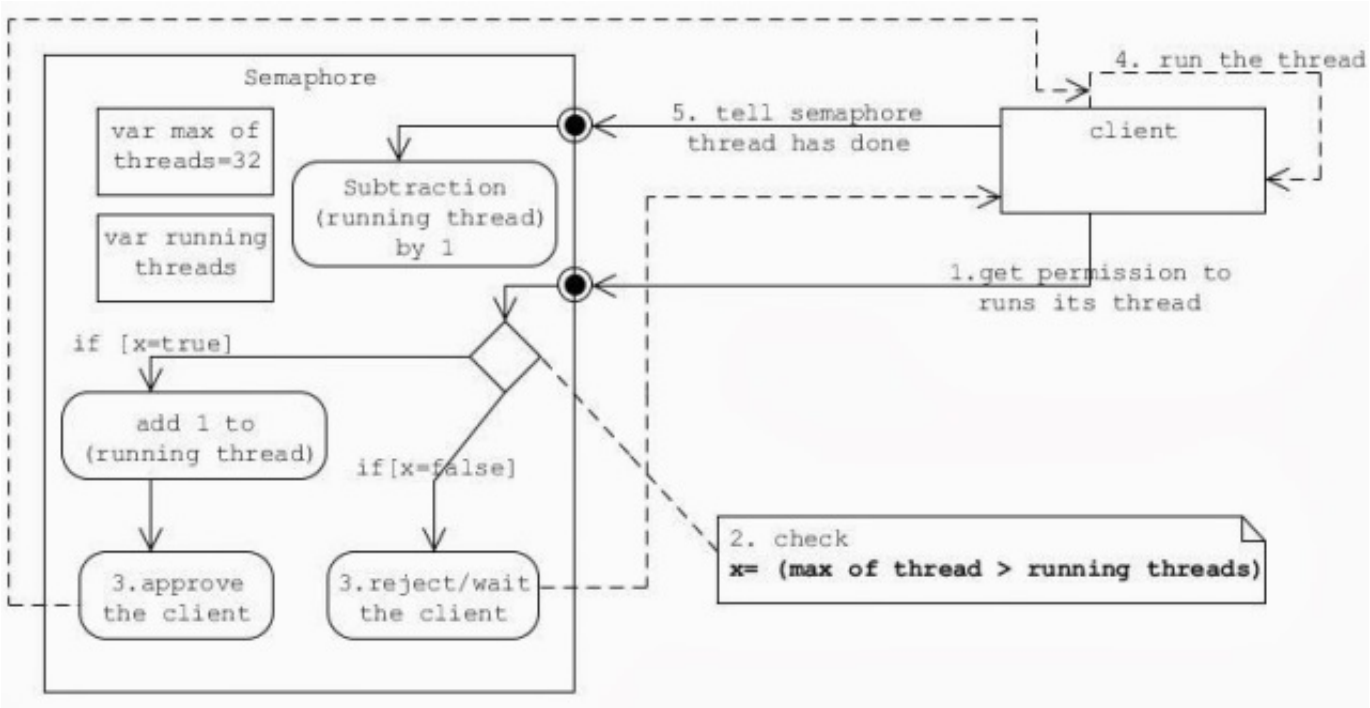
Note: if you ask me, and other guys who know multi-thread programming, we say 5000 thread in an application is not a dragon killing story(as you may feel), because you would run over 100000 threads on a GPU, 5000 thread is a little hard to manage for the CPU(also manage the memory), so in future you have to separate your application into concurrent and serial modules (execution flow), and separate the code between GPU and CPU (not so hard, keep cool).

Semaphore, Limiting Maximum of Running Threads

Well one of the patterns which causes limiting the maximum of running thread is called (Semaphore), this is very useful to prevent your application of running to much threads that would causes memory and process(execution-time) issues, here semaphore ensures you, there isn't going to be more than n maximum thread running.

This is useful when your application is small (not recommended for big applications). There is a good implemented semaphore class available in its Java SE API, but I suggest you do it yourself, if you want to know what a semaphore really does!?

For having a semaphore, you have to know what does it do exactly, well semaphore looks like a execution permission system, at time t0 you want to run a thread, so first you have to get the permission of semaphore. Here semaphore tracks every running threads and decides to either approves, blocks or rejects you. this is going to be easy.



The above semaphore tracks the running threads (in-direct) with counting, note that here clients have to signal semaphore again and tell it that has finished their work. Then a value removed by the running threads in semaphore. this kind of semaphore is known as **counting semaphore**, and now let's have a simple impl of this one.

```
package arash.blogger.example.thread;
/**
   * by Arash M. Dehghani
   * arashmd.blogspot.com
   */
public class Core {
    public static void main(String[] args) throws InterruptedException {
        System.out.print("trying to run 300 threads, but max 16 thread at a time\n");
        MySemaphore sem=new MySemaphore(16);//it could be 8, 32, or even 32, dependent to your
application business
        for(int i=0;i<300;i++){
            sem.acquire();//get the permission to run a thread [1]
            new Thread(new MyThread(sem,i)).start(); //runs the thread [3]
        }
        Thread.sleep(100);
        System.out.print("Completed :)\n");
    }
}

class MyThread implements Runnable {
    MySemaphore sem;int val;
    public MyThread(MySemaphore sem,int val){
        this.sem=sem;//pass the semaphore to client run, to release itself after it get
terminated
        this.val=val;
    }
    @Override
    public void run() {
        try{
        System.out.print("Hello from client run\n client No: "+this.val+"\n");
        Thread.sleep(1000);
        }catch(Exception e){e.printStackTrace();}
        finally{
            sem.release(); //tells the semaphore that it has finished its job [4]
        }
    }
}

class MySemaphore{
    int maxThreadSize;
    int runningThreadSize=0;
    Object lock=new Object(),elock=new Object();
    public MySemaphore(int maxThreadSize){
        this.maxThreadSize=maxThreadSize;
    }
    public synchronized void acquire() throws InterruptedException{
        if(runningThreadSize==maxThreadSize){
                this.wait();//checks is it possible to run a thread at this time?
                //or should wait until another thread get finished [2]
            }
        runningThreadSize++;
    }
    public synchronized void release(){
            runningThreadSize--;//decrease one thread from runningThreadSize
            try{
            this.notify();//notify any waiting thread (if any) [5]
            }catch(IllegalMonitorStateException e){
                //if no any thread were not waiting for this, then this block get caught
            }
    }
}
```

The above example is about running 300 thread, that this is not a good idea run every 300 threads at a time, we want to limit it(at a time) with semaphore.

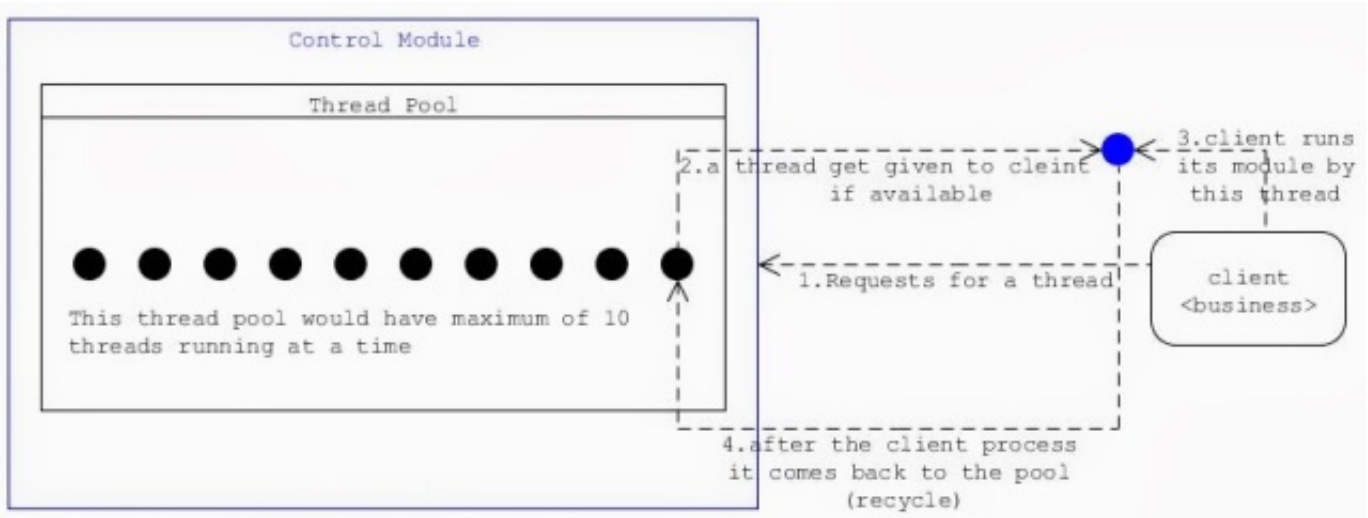
The main thread gets permission for each 300 thread wants to run, and semaphore has to decide to either wait or approve the request, if maximum of threads are running, then semaphore blocks until one of the threads(running threads) get terminated and signal the semaphore, so semaphore ensures that one thread got finished, and it's possible to run a new one. Simply, you would reject (throw an exception, error, false,...) the main thread instead of block it.

Recyclable Threads (Thread Pool)

As we mentioned, a thread has some properties(takes some memory), but there is one important thing belong them, OS and JVM don't care about what's going on (execution flow) in a thread, they know just that this thread should get run, this is like a postman, it brings every envelop to destinations and doesn't care what is inside, then the envelop packet get destroyed, and a new packet should get used for the new one, so here its time-consuming about destroying and creating a new packet, all want I say that, why don't use the old packet for the next envelop too? or why don't use the existing thread for the next thread too?

But when we implement a thread (runnable) that copies two files in a flash disk, then this thread just copies two files in a flash disk forever, here we have to impl our thread in such a way that independent to any business flow, a thread should be a thread, not my business!

Well, this could be done very easy by changing the impl of your application, also you need to impl a module which controls these threads and execution flows, it could be like this.



As the above figure says, a thread is got from the thread pool, it does something and has to back to the pool for the next request. so here the thread object should independent of all businesses, that is really easy to solve, when we pass the business as a interface to the thread, like this.

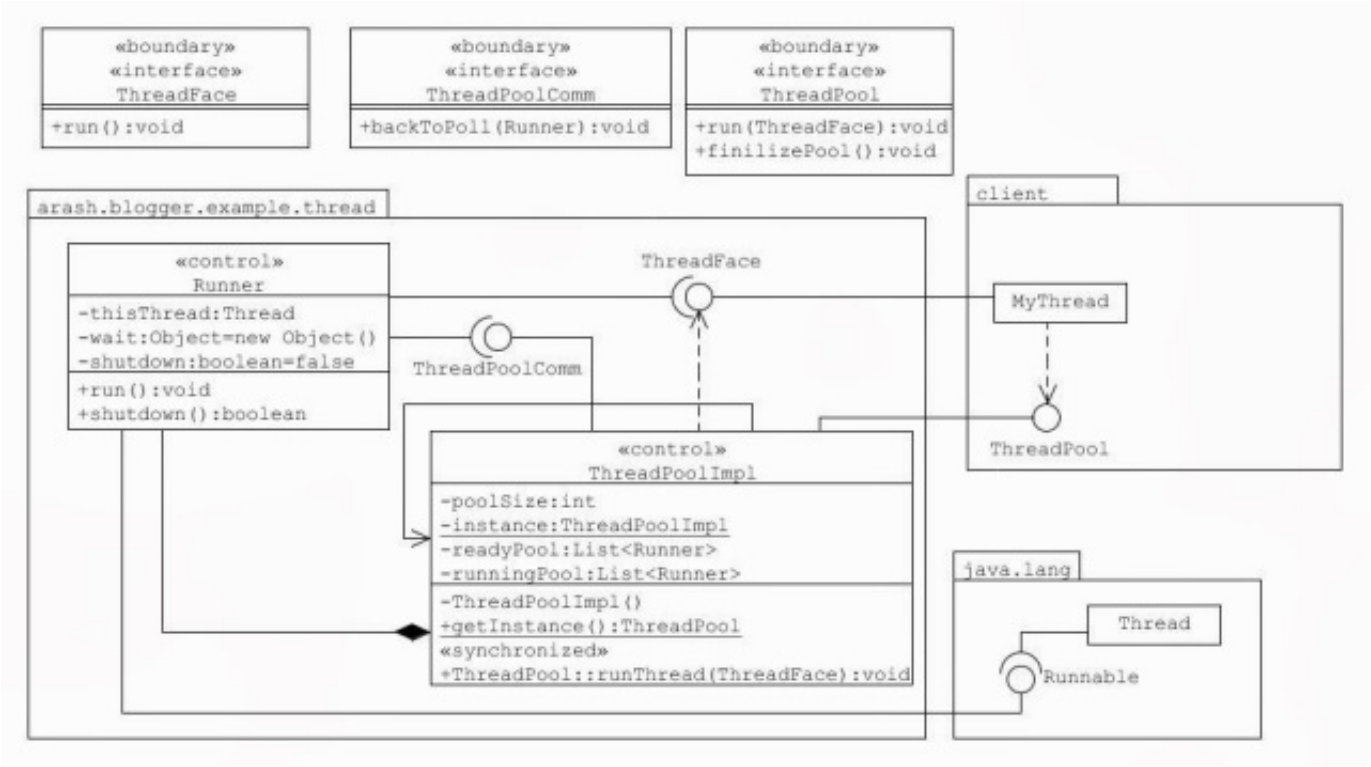
```
public interface ThreadFace{
   public void run();//it could return something too, dependent on your design
}
//
//file Runner.java
public final class Runner implements Runnable{//NOTE: incomplete code!
  private ThreadFace tf;//contains the target business, that should get run (is provided by
client)
  @Override
  public void run(){
       tf.run();//runs the client business code
```

```
  }
}
```

and more detailed code:

```
//file ThreadFace.java
public interface ThreadFace{
   public void run();//it could return something too, dependent on your design
}
//
//file Runner.java
public final class Runner implements Runnable{//NOTE: incomplete code!
  private Thread thisThread;
  private Object wait=new Object();
  private boolean shutdown=false;
  private ThreadFace tf;//contains the target business, that should get run (is provided by
client)
  void runIt(ThreadFace tf){this.tf=tf;
    synchronized(wait){wait.notify();}//signal the run(actual thread) that a new job has
arrived, run it
  }
  @Override
  public void run(){
    this.thisThread=Thread.currentThread();
    while(!shutdown){
       synchronized (wait){ wait.wait();//wait until this thread gets a job}
       tf.run();//runs the client business code
       this.sendFinSignal();
    }
  }
  private void sendFinSignal(){
    //tells thread pool that this job has been finished, take me back to pool
  }
  void shutdown(){//thread pool signal this thread for shutting
  // down itself completely (application should get shutdown)
    this.shutdown=true;
    this.thisThread.interrupt();//interrupt the current thread
  }
}
//
```

The above code is just part of a simple thread pool, we haven't designed thread pool yet, the following class diagram is for guys who can understand UML, very simple.



It contains the members we need for our thread pool, let me start with interfaces, as you see there are three(3) interfaces is used in our design, so let's have a closer look to them, what are they for?

ThreadFace

This interface is used as a bridge between actual thread and the virtual thread, here virtual thread points to a method that should get run as a thread, but it (ThreadFace) is not a real thread(Runnable), well it will get run by the real thread in Runner class, it contains a void method which does not accept any argument, as same as actual Runnable interface, in fact the actual thread will run the ThreadFace's run method.

```
interface ThreadFace{
 public void run();//looks like the Runnable, but could be in another face too
}
```

ThreadPool

This interface is used for communication between client and thread pool, client needs to pass ThreadFace's (implementation) instance to the ThreadPool, then actual ThreadPool (ThreadPoolImpl) has a decision to either runs or waits the client until a new ready thread.

As the previous diagram says, client sends its request (by ThreadFace) to the ThreadPool (ThreadPoolImpl), and if there is an idle thread available, then thread will run, else client has to wait (blocked by pool) until a thread finishes its job and returns to the pool and does the client job. also because of actual threads (Runner) are immortal(in a true while loop) so this interface provides another method for shutting down every threads.

```
interface ThreadPool{
 public void run(ThreadFace tf)throws InterruptedException;//for running something
 public void finilizePool();//for shutting down the pool
}
```

ThreadPoolComm

This interface is used for communicating between actual threads (Runner) and the thread pool (ThreadPoolImpl), whenever a thread finishes its job, it acquaints the thread pool with this interface that its job has finished and is ready to back to the pool.

```
interface ThreadPoolComm{
 public void backToPool(Runner r);//for signal and heading back to the pool
}
```

Okay, and now time for classes.

Runner

I would call it actual thread, this class implements the Runnable interface, and it means it could be a thread, this class just has communication with ThreadPoolComm and ThreadFace interfaces, inside the run method it runs the ThreadFace interface passed by ThreadPoolImpl class, so because it runs a interface, it could be anything, it could have any implementation. this class gets run its thread (run) by the ThreadPool, and waits for a signal for the very next ThreadFace object, after it notified by pool, it runs the ThreadFace's runs method and signals the pool to head back to the thread pool again for the next request, this class is hand of our thread pool application.

```
final class Runner implements Runnable{
 private Thread thisThread;//for storing the current(this) thread
 public Runner(ThreadPoolComm tpc){this.tpc=tpc;}//get the pool communicator interface by
constructor
 private ThreadPoolComm tpc;
  private Object wait=new Object();
  private boolean shutdown=false;
  //there is no any read and update at same time situation, so it's could be not volatile

  private ThreadFace tf;//contains the target business, that should get run (is provided by
client)
  public void runIt(ThreadFace tf){this.tf=tf;//for any request, thread pool calls this method
    synchronized(this.wait){this.wait.notify();}
    //signal the run(actual thread) that a new job has arrived, run it! [2]
  }
  @Override
  public void run(){
   this.thisThread=Thread.currentThread();//getting current(this) thread
     while(shutdown==false){//while it has to roll!
      try{
       synchronized (this.wait){
        this.wait.wait();//wait until this thread gets a job [1]
        tf.run();//runs the client business code  [3]
        }
       }catch(Throwable e){
          //logging, any alternative operation, etc... [b]
         }
       this.sendFinSignal();//heading back to the pool [4]
     }
     System.out.print("A Runner thread wrapper has stopped\n");// [c]
    }
  private void sendFinSignal(){
   tpc.backToPool(this);
```

```
    //tells thread pool that this job has been finished, take me back to pool
  }
  void shutdown(){//thread pool signal this thread for shutting
  // down itself completely (application should get shutdown)
    this.shutdown=true;//set the shutdown flag to true
    this.thisThread.interrupt();//interrupt the thread [a]
  }
}
```

ThreadPoolImpl

If Runner class is the hand of the application, then ThreadPoolImpl is the heart of our application, this class manages everything, it gets the ThreadFace object by the client and passes it to Runner to run it, it also generates and manages the Runner instances too, it also sends the shutdown signal to every running/ready threads. Whenever a client needs to run its ThreadFace, this class checks is there any ready thread or not? if yes, then pass the ThreadFace object to the Runner object and job get started, if no, then it blocks the client until a running thread (Runner) finishes its job and heads back to the pool, then pass it to it. this class contains two list of Runner class, readyPool and runningPool, first one contains idle(ready) threads that are ready for action, and second one contains threads that are doing something(busy).

There is a note belong to this class, when we use thread pool, we want to limit the maximum running thread at a time, and also recycle the threads too, so there is only one instance of this class is required, for this we hide the constructor, so no one would create another instance.

But we need at least one, it could be done by one static ThreadPoolImpl instance, in fact this class, itself creates one instance of itself and stores inside, and clients would get this instance by a static method and work with it, this is a simple pattern, called Singleton, very useful!

```
public class ThreadPoolImpl implements ThreadPool, ThreadPoolComm{

  public static void main(String[] args) throws InterruptedException{
      ThreadPool pool=ThreadPoolImpl.getInstance();
      for(int i=0;i<1024;i++){
          pool.run(new Client(i));
          Thread.sleep(10);
      }
      Thread.sleep(2000);
      pool.finilizePool();
  }
  private int poolSize=32;//the size of the pool (threads), it could be anything
  private static ThreadPoolImpl instance=new ThreadPoolImpl();// let's have a instance inside,
just one!
  private List<Runner> readyPool,runningPool;//two list of Runners(threads), ready ones, and
busy ones
  public static ThreadPool getInstance(){//get the instance with this guy
      return ThreadPoolImpl.instance;
  }
  private ThreadPoolImpl(){//hide the constructor, so no one will be able to create one
      System.out.print("Thread Pool Initializing....\n");
      readyPool=new ArrayList<>(poolSize);
      runningPool=new ArrayList<>(poolSize);
      for(int i=0;i<poolSize;i++){//filling the pool
          Runner r=new Runner(this);//pass the ThreadPoolComm to the thread(this)
          new Thread(r).start();// starts each thread by their creation
          readyPool.add(r);//add the ready thread to ready pool
      }
      System.out.print("Thread Pool initialized with "+poolSize+" threads\n");
  }
```

```
  @Override
  public synchronized void run(ThreadFace tf) throws InterruptedException {
      if(readyPool.size()==0){//checks, is there any available thread? [1]
      //  System.out.print("no any Threads available, wait!\n");
          this.wait();//if no, so wait until one heads back
          }Runner r=readyPool.get(0);//get the first thread object from the list [2]
          readyPool.remove(0);//remove the thread from the ready pool [3]
          runningPool.add(r);//add it to busy pool [4]
          r.runIt(tf);//signal it, to run the clients code [5]
          //System.out.print("Thread run\n");
  }
  @Override
  public synchronized void backToPool(Runner r) {//called when a thread finishes its job
          readyPool.add(r);//add the thread to ready pool [6]
          runningPool.remove(r); //remove the thread from the busy thread [7]
      try{
          this.notify();//notify  waiting thread(if any) that a ready thread is available [0=8]
      }catch(Exception e){}
  }
  @Override
  public synchronized void finilizePool() {
      for(Runner r : readyPool){
          r.shutdown();//send shutdown to every ready threads
      }
      for(Runner r : runningPool){
          r.shutdown();//send shutdown to every running threads
      }
  }

}
```

Client

It could be anything, it dependents on the business, but here we have a very very simple one. note that a client should implements the ThreadFace, so we could call it our client.

```
class Client implements ThreadFace {
 public Client(int i) {
  this.i = i;
 }
 int i;
 @Override
 public void run() {
  System.out.print("Hello! thread no:" + i + " \n");
  try {
   Thread.sleep(1000);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
 }
}
```

Enough talking, let's have a look at the whole code!

```
package arash.blogger.example.thread;

import java.util.ArrayList;
import java.util.List;
/**
```

```
 * by Arash M. Dehghani
 * arashmd.blogspot.com
 */
public class ThreadPoolImpl implements ThreadPool, ThreadPoolComm {

  public static void main(String[] args) throws InterruptedException {
    ThreadPool pool = ThreadPoolImpl.getInstance();
    for (int i = 0; i < 1024; i++) {
      pool.run(new Client(i));
      Thread.sleep(10);
    }
    Thread.sleep(2000);
    pool.finilizePool();
  }
  private int poolSize = 32;// the size of the pool (threads), it could be
                            // anything
  private static ThreadPoolImpl instance = new ThreadPoolImpl();// let's have a
                                                                // instance
                                                                // inside, just
                                                                // one!
  private List<Runner> readyPool, runningPool;// two list of Runners(threads),
                                              // ready ones, and busy ones
  public static ThreadPool getInstance() {// get the instance with this guy
    return ThreadPoolImpl.instance;
  }
  private ThreadPoolImpl() {// hide the constructor, so no one will be able to
                            // create one
    System.out.print("Thread Pool Initializing....\n");
    readyPool = new ArrayList<Runner>(poolSize);
    runningPool = new ArrayList<Runner>(poolSize);
    for (int i = 0; i < poolSize; i++) {// filling the pool
      Runner r = new Runner(this);// pass the ThreadPoolComm to the thread(this)
      new Thread(r).start();// starts each thread by their creation
      readyPool.add(r);// add the ready thread to ready pool
    }
    System.out.print("Thread Pool initialized with " + poolSize + " threads\n");
  }
  @Override
  public synchronized void run(ThreadFace tf) throws InterruptedException {
    if (readyPool.size() == 0) {// checks, is there any available thread? [1]
      // System.out.print("no any Threads available, wait!\n");
      this.wait();// if no, so wait until one heads back
    }
    Runner r = readyPool.get(0);// get the first thread object from the list [2]
    readyPool.remove(0);// remove the thread from the ready pool [3]
    runningPool.add(r);// add it to busy pool [4]
    r.runIt(tf);// signal it, to run the clients code [5]
    // System.out.print("Thread run\n");
  }
  @Override
  public synchronized void backToPool(Runner r) {// called when a thread
                                                 // finishes its job
    readyPool.add(r);// add the thread to ready pool [6]
    runningPool.remove(r); // remove the thread from the busy thread [7]
    try {
      this.notify();// notify waiting thread(if any) that a ready thread is
                    // available [0=8]
    } catch (Exception e) {
    }
  }
  @Override
  public synchronized void finilizePool() {
```

```
    for (Runner r : readyPool) {
      r.shutdown();// send shutdown to every ready threads
    }
    for (Runner r : runningPool) {
      r.shutdown();// send shutdown to every running threads
    }
  }

}
interface ThreadPool {
  public void run(ThreadFace tf) throws InterruptedException;
  public void finilizePool();
}
interface ThreadFace {
  public void run();
}
interface ThreadPoolComm {
  public void backToPool(Runner r);
}
final class Runner implements Runnable {
  private Thread thisThread;// for storing the current(this) thread
  public Runner(ThreadPoolComm tpc) {
    this.tpc = tpc;
  }// get the pool communicator interface by constructor
  private ThreadPoolComm tpc;
  private Object wait = new Object();
  private volatile boolean shutdown = false;// there is no any read and update at same
                                    // time situation, so it's could be not
                                    // volatile
  private ThreadFace tf;// contains the target business, that should get run (is
                        // provided by client)
  public void runIt(ThreadFace tf) {
    this.tf = tf;// for any request, thread pool calls this method
    synchronized (this.wait) {
      this.wait.notify();
    }// signal the run(actual thread) that a new job has arrived, run it! [2]
  }
  @Override
  public void run() {
    this.thisThread = Thread.currentThread();// getting current(this) thread
    while (shutdown == false) {// while it has to roll!
      try {
        synchronized (this.wait) {
          this.wait.wait();// wait until this thread gets a job [1]
          tf.run();// runs the client business code [3]
        }
      } catch (Throwable e) {
        // logging, any alternative operation, etc... [b]
      }
      this.sendFinSignal();// heading back to the pool [4]
    }
    System.out.print("A Runner thread wrapper has stopped\n");// [c]
  }
  private void sendFinSignal() {
    tpc.backToPool(this);
    // tells thread pool that this job has been finished, take me back to pool
  }
  void shutdown() {// thread pool signal this thread for shutting down itself
                   // completely (application should get shutdown)
    this.shutdown = true;// set the shutdown flag to true
    this.thisThread.interrupt();// interrupt the thread [a]
  }
```

```
}
class Client implements ThreadFace {
  public Client(int i) {
    this.i = i;
  }
  int i;
  @Override
  public void run() {
    System.out.print("Hello! thread no:" + i + " \n");
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Just run the above code, and monitor your system, the above code runs very fast, because 32 threads for CPU is not to much hard to execute (at least better than 5000 at the same time), also the most important thing is that a thread gets used many times, there is no time needed to destroying and creating threads for every thread and this is the main advantage over semaphore.

Well, I'm planning to writing another article about java threading(this one is big enough almost) but in advance, also mention about high performance computing (HPC) in Java too, so if you found this article good enough, just keep tight with updates. also you can find another example here and once again, for any issues, questions, problems, just feel free to contact me, have a good concurrent program.

License

About the Author

Arash M. Dehghani

Unknown

No Biography provided

Comments and Discussions

15 messages have been posted for this article Visit **http://www.codeproject.com/Articles/616109/Java-Thread-Tutorial** to post and view comments on this article, or click **here** to get a print view with messages.