Java Programming: Concurrent Programming in Java

Manuel Oriol

May 10, 2007

1 Introduction

Threads are one of the fundamental structures in Java. They are used in a lot of applications as their use is meant to be simple. In practice, very few is required from programmer's perspective. It is only needed to simply specialize a class or define a method to benefit from the Thread facility.

When considering concurrent programs, there is a need to synchronize. As an introduction, let precise that each object has a lock that can be used to synchronize threads. The final methods wait, notify and notifyAll are all manipulating this lock as well as synchronized methods and blocks.

Section 2 presents threads and runnable objects. Section 5 explains how low-level mechanisms work. Section 4 presents synchronized blocks and methods. Section 3 presents volatile variables.

2 Threads, Runnables and ThreadGroup

2.1 Threads and Runnable

The class Thread is the primary class for defining threads. By calling the start() method on a thread, programmers create a new thread that executes the run() method. Once the thread has been launched, the start() method returns. Note that thread can only be started once. In particular, they cannot be restarted even if they finished execution.

The simplest way to create a new Thread is thus to subclass the class Thread and override the method run() and include the code that one wants to execute.

As shown in Table 1 Instances of the class Thread can also be created by providing instances of classes that implement interface Runnable. The interface itself only requires to have the run() method to be implemented. As an example, it is possible to create a new thread using an object o that is an instance of a class implementing Runnable:

Runnable o;
...
(new Thread(Runnable)).start();
Table 1: Thread Methods

<table>
<thead>
<tr>
<th>Method (javadoc)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread()</td>
<td>Creates a new thread.</td>
</tr>
<tr>
<td>Thread(Runnable target)</td>
<td>New thread with code of the argument.</td>
</tr>
<tr>
<td>Thread(Runnable target, String name)</td>
<td>New thread with a name and custom code.</td>
</tr>
<tr>
<td>Thread(String name)</td>
<td>New thread with a name.</td>
</tr>
<tr>
<td>Thread(ThreadGroup group, Runnable target)</td>
<td>Creates a new thread and adds it to a specific ThreadGroup.</td>
</tr>
<tr>
<td>Thread(ThreadGroup group, Runnable target, String name)</td>
<td>Creates a new thread with a given name and adds it to a specific ThreadGroup.</td>
</tr>
<tr>
<td>Thread(ThreadGroup group, Runnable target, String name, long stackSize)</td>
<td>Creates a new thread with a given name and adds it to a specific ThreadGroup. The stack size is indicative.</td>
</tr>
<tr>
<td>Thread(ThreadGroup group, String name)</td>
<td>Creates a new thread with a given name and adds it to a specific ThreadGroup.</td>
</tr>
<tr>
<td>static Thread currentThread()</td>
<td>Returns a reference on current thread.</td>
</tr>
<tr>
<td>String getName()</td>
<td>Returns this thread name</td>
</tr>
<tr>
<td>Thread.State getState()</td>
<td>Returns this thread state (interrupted...)</td>
</tr>
<tr>
<td>int getPriority()</td>
<td>Returns this thread priority</td>
</tr>
<tr>
<td>ThreadGroup getThreadGroup()</td>
<td>Returns this thread’s group</td>
</tr>
<tr>
<td>void interrupt()</td>
<td>Inturrupts the thread.</td>
</tr>
<tr>
<td>static boolean interrupted()</td>
<td>Checks if the current thread has been interrupted.</td>
</tr>
<tr>
<td>void join()</td>
<td>Waits for the thread to end</td>
</tr>
<tr>
<td>void join(long millis)</td>
<td>Waits at most millis milliseconds for this thread to end.</td>
</tr>
<tr>
<td>void run()</td>
<td>Method called upon thread starting.</td>
</tr>
<tr>
<td>static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)</td>
<td>Changes the exception handler for this thread.</td>
</tr>
<tr>
<td>static void sleep(long millis)</td>
<td>Stops the current thread for millis milliseconds.</td>
</tr>
<tr>
<td>void start()</td>
<td>Starts a new thread.</td>
</tr>
</tbody>
</table>

2.2 ThreadGroup

Threads are grouped in ThreadGroups. ThreadGroups are hierarchical: each one can contain one or more other ThreadGroups. Each ThreadGroup is an execution domain that contains the threads. In particular, a thread belonging to a ThreadGroup can only modify/ask information about its ThreadGroup’s threads or the threads of the children of its ThreadGroup (and recursively). By default, it is not possible to interrupt threads from another ThreadGroup.

It is possible to override this feature by setting up different policies through the SecurityManager.
2.3 Interrupting a Thread

What is argued in the APIs is that the methods `destroy()`, `stop()`, `suspend()` and `resume()` are deprecated meaning that they should be avoided if possible and that they may be removed at any time when the language evolves.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void destroy()</code></td>
<td>Not implemented. Returns an exception.</td>
</tr>
<tr>
<td><code>void resume()</code></td>
<td>Resumes execution of a suspended thread.</td>
</tr>
<tr>
<td><code>void stop()</code></td>
<td>Stops the execution. Locks are released.</td>
</tr>
<tr>
<td><code>void stop(Throwable obj)</code></td>
<td>Stops the thread and raise the throwable.</td>
</tr>
<tr>
<td><code>void suspend()</code></td>
<td>Stops the thread and suspend the thread</td>
</tr>
</tbody>
</table>

Table 2: Thread Deprecated Methods

The solution offered¹ states that calls to `stop()` and `suspend()` should be avoided to use `interrupt()`. The method itself only sets a flag returned by `interrupted()` to `true` and visits the `interruptible` operations (visit APIs² for more details).

In practice even if the deprecated methods are unsafe, they are the only real way to interrupting threads and they are implemented(with the notable exception of `destroy`). The only precaution to take when using them, is to ensure that locks are released or threads are not interrupted while others need to release locks.

3 volatile variables

`volatile` variables are variables on which operations are executed in an atomic manner. It is also stated in the Java 5.0 specifications that operations on `volatile` variables respect sequential consistency: programmers can be sure that they execute in the same order as they were specified (or at least it is not possible to observe a behavior that would be invalidate with respect to that assumption);

4 synchronized blocks/methods

To synchronize blocks of instructions, the best way is to use the `synchronized` keyword. This keyword can be applied to both a block of code and to a method body. In any case, if the lock to be taken is already taken, the code blocks until the lock is released. When the lock is released, there is a competition between all waiters and only one can grab the lock.

When applied to a method body, the method invocation is taking the lock on the object on which the method is applied. If the method is a class method

¹http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html
²http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html#interrupt()
(static) the lock is taken on the class object. As an example, a synchronized method can be declared as follows:

```java
synchronized void incCount()
{   
    count3++;
}
```

Another way of taking synchronization locks is to use the `synchronized` keyword on an object to take its lock. The block following the keyword is then having the lock. As an example, an equivalent form of the proceeding example is:

```java
void incCount()
{
    synchronized(this){
        count3++;
    }
}
```

5 wait and notify

As it is often needed to synchronize multiple threads at the same time, there are other constructs that allow synchronization. In particular, while in a synchronized block, it is possible to release a lock and wait for a notification to wake-up and try to grab the lock. This is typically realized through the use of the `wait()` method.

In order to wake up processes waiting on a given lock, it is needed to use `notify()` or `notifyAll()` to respectively wake up one waiting thread or all waiting threads. This can also be done only within a block that has the lock on the considered object. When woken up, threads try to grab the lock on which they have been notified and block until they grab it.

As an example let consider two threads that perform a ping-pong sequence:

```java
public class ThreadExample2 {
    private static class MyThread extends Thread{
        boolean isPing;
        Object lock;
        private MyThread(Object lock, boolean isPing){
            this.lock=lock;
            this.isPing=isPing;
        }
        public void pingPong()throws Exception{
            synchronized(lock){
                while(true){
                    if (isPing)
                        System.out.println("Ping");
                    else
                        System.out.println("Pong");
                    lock.notifyAll();
                    lock.wait();
                }
            }
    }
    public class ThreadExample2 {
    private static class MyThread extends Thread{
        boolean isPing;
        Object lock;
        private MyThread(Object lock, boolean isPing){
            this.lock=lock;
            this.isPing=isPing;
        }
        public void pingPong()throws Exception{
            synchronized(lock){
                while(true){
                    if (isPing)
                        System.out.println("Ping");
                    else
                        System.out.println("Pong");
                    lock.notifyAll();
                    lock.wait();
                }
            }
    }
    ```
6 Exercises

1. Is the following program going to terminate?

```java
public class ThreadExample {
    private static class MyThread extends Thread{
        private static int count;
        private static volatile int count2;
        private static int count3;
        Object lock;
        private MyThread(Object lock){
            this.lock=lock;
        }
        public void run(){
            synchronized(lock){
                lock.notifyAll();
                count++;
                try {
                    lock.wait();
                } catch (Exception e){}
            }
        }
        synchronized static void incCount(){
            count3++;
        }
    }
    public static void main(String[] args){
        Object o=new Object();
        (new MyThread(o,true)).start();
        (new MyThread(o,false)).start();
    }
}
```

2. Is it possible that the following program have the given trace?

```java
5
```
public class ThreadExample {
    private static class MyThread extends Thread{
        private static int count;
        private static volatile int count2;
        private static int count3;
        Object lock;
        private MyThread(Object lock){
            this.lock=lock;
        }
        public void run(){
            synchronized(lock){
                count++;
            }
            count2++;
            incCount();
            System.out.println("count=\n count2=\n count3=\n");
        }
        synchronized static void incCount(){
            count3++;
        }
    }
    public static void main(String[] args){
        Object o=new Object();
        for (int i=0; i<10;i++)
            (new MyThread(o)).start();
    }
}

Trace:

3  count=3 count2=4 count3=4
4  count=4 count2=4 count3=4
8  count=8 count2=8 count3=8
9  count=9 count2=9 count3=9
5  count=5 count2=5 count3=5
10 count=10 count2=10 count3=10
7  count=7 count2=7 count3=7
8  count=8 count2=8 count3=8
6  count=6 count2=6 count3=6
9  count=9 count2=9 count3=9

3. A thread pool is a group of threads that can be used to program a server program without creating new threads each time a new connection is created but rather reuse existing ones. How would you program a thread pool?