Introduction to multithreading in Java

Steven Latré, Jeroen Famaey and Tom De Schepper – University of Antwerp
Tim Verbelen - Ghent University

2016 - 2017

1 Introduction

The goal of this tutorial is to familiarise yourself with the concept of multithreading. In this tutorial we will explore how to create and synchronize multiple threads in Java.

2 Basic Threading

In concurrent programming, there are two basic units of execution: processes and threads. In the Java programming language, concurrent programming is mostly concerned with threads. A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space. Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. Most implementations of the Java virtual machine run as a single process. Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process. Threads exist within a process every process has at least one. Threads share the process resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

2.1 Defining and starting a thread

In Java each thread is associated with an instance of the class `Thread`. An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

- Provide a Runnable object. The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:

```
public class HelloRunnable implements Runnable {
    public void run () {
        System.out.println("Hello from a thread!");
     }

    public static void main(String[] args) {
        Thread t = new Thread (new HelloRunnable ());
        t.start();
    }
}
```

- Subclass Thread. The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

```
public class HelloThread extends Thread {
    public void run () {
        System.out.println("Hello from a thread!" ) ;
    }

    public static void main (String args []) {
        Thread t = new HelloThread ();
        t.start();
    }
}
```

Notice that both examples invoke Thread.start() in order to start the new thread. **It is important to note that you should not directly invoke the run method, as this will execute the code from within the current thread.**

Which of these idioms should you use? The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread. The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread. In the remainder we will use the first approach, since it is more flexible and separates the Runnable task from the Thread object that executes the task.

2.2 Pausing execution with sleep

Thread.sleep() causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.

Two overloaded versions of sleep are provided: one that specifies the sleep time in millisecond and one that specifies the sleep time in nanosecond. However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS. Also, the sleep period can be terminated by interrupts, as we will see in the next section. In any case, you cannot assume that invoking sleep will suspend the thread for precisely the time period specified.

```
try{
    Thread.sleep(4000);
} catch(InterruptedException e){ }
```

Notice that we catch InterruptedException. This is an exception that sleep throws when another thread interrupts the current thread while sleep is active.

2.3 Interrupts

An interrupt is an indication to a thread that it should stop what it is doing and do something else. It is up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

A thread sends an interrupt by invoking the interrupt method on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption. It can support interruption in two ways. If the thread frequently calls methods that throw InterruptedException, it simply returns from the run method after it catches that exception. If a thread goes a long time without invoking a method that throws InterruptedException, then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received.

```
for (int i = 0; i < inputs.length ; i++) {
    heavyCrunch (inputs[i]);
    if(Thread.interrupted()){
        //We've been interrupted no more crunching
        return;
     }
}
```

2.4 Joins

The join method allows one thread to wait for the completion of another. If t is a Thread object that is currently running,

```
t.join();
```

causes the current thread to pause execution until thread t terminates. Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify. Like sleep, join responds to an interrupt by exiting with an InterruptedException.

2.5 Example

Below is an example to illustrate the concepts described above. The Greeter thread prints out greeting messages with a sleep interval of one second in between. The main thread starts up the greeter and waits until the Greeter thread is finished using the join method.

```
package threading;

public class Greeter implements Runnable {

        String[] greetings = { "Hallo!", "Guten Tag!",
            "Bonjour!", "Hello!" };

        public void run() {
                for (int i = 0; i < greetings.length; i++) {
                        System.out.println(greetings[i]);
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException ex) {
                                return;
                        }
                }
        }

        public static void main(String[] args) {
                Thread t = new Thread(new Greeter());
                t.start();
                try {
                        t.join();
                } catch (InterruptedException e) {
                        return;
                }
                System.out.println("Greeter completed!");
        }
}
```

3 Working in parallel

One of the reasons to use threads is to improve the performance by executing tasks in parallel. On multi-core architectures processing intensive tasks can be sped up significantly, but even on single core architectures, the performance can be increased by executing I/O-operations in parallel with the processing. For example, read a DVD from disk and in parallel transcode the movie to a format suitable for your portable media player.

Consider the following class Counter which maintains a long as a counter which can be incremented and decremented through the increment() and decrement() method. Next we have an Incrementor thread that will increment the Counter a number of times. The Main class will spawn a number of Incrementor threads all willing to increment the Counter object.

Counter class

```
package counter;

public class Counter {
```

```
        private long value;

        /**
         * Constructor with the initial value of the counter
         *
         * @param startValue
         *            the initial value
         */
        public Counter(long startValue) {
                this.value = startValue;
        }

        /**
         * Increment the counter
         */
        public void increment() {
                value++;
        }

        /**
         * Decrement the counter
         */
        public void decrement() {
                value--;
        }

        /**
         * Get the cur rent value of the counter
         *
         * @return the cur rent counter value
         */
        public long getValue() {
                return value;
        }

        /**
         * Get the string representation of this Counter
         *
         * @return a String representing this counter
         */
        @Override
        public String toString() {
                return "" + value;
        }
}
```

Incrementor class

```
package counter;

public class Incrementor implements Runnable {

        private Counter counter;
        private long count;
```

```
        /**
         * Constructor with the counter to use and the number of iterations
         *
         * @param counter
         *            The Counter to use for processing
         * @param count
         *            The number of increment operations to perform.
         */
        public Incrementor(Counter counter, long count) {
                this.counter = counter;
                this.count = count;
        }

        /**
         * Performing the actual business logic .
         */
        public void run() {
                for (int i = 0; i < count; i++) {
                        counter.increment();
                }

                System.out.println("Incrementor finished");
        }
}
```

Main class

```
package counter;

import java.util.ArrayList;
import java.util.List;

public class Main {

        public static void main(String[] args) {
                Counter counter = new Counter(0);

                long countLimit = 100000000;
                int threadCount = 15;

                // Print the expected result from the code
                System.out.println(countLimit * threadCount);
                List<Thread> threads = new ArrayList<Thread>();

                for (int i = 0; i < threadCount; i++) {
                        Thread t = new Thread(new Incrementor(counter, countLimit));
                        threads.add(t);
                        t.start();
                }
                for (int i = 0; i < threadCount; i++) {
                        try {
                                threads.get(i).join();
                        } catch (InterruptedException ex) {
```

```
                }
            }
            // Print the obtained result
            System.out.println(counter);
        }
}
```

When you run this sample application you will notice that when the countLimit is large enough the output of the program is not the correct value. Because improper thread synchronization the following situation can occur between two Incrementors A and B:

1. Incrementor A : retrieve Counter value (e.g. 10)
2. Incrementor B : retrieve Counter value (still 10)
3. Incrementor A : increment retrieved value (result is 11)
4. Incrementor B : increment retrieved value (result is also 11)
5. Incrementor A : store result in value (value is now 11)
6. Incrementor B : store result in value (value is 11)

As you can see one incrementation is lost due to improper thread synchronization. In order to synchronize threads, Java provides two basic synchronization idioms: synchronized methods and synchronized statements.

3.1 Synchronized methods

To make a method synchronized, simply add the synchronized keyword to its declaration. This has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block suspend execution until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

```
package synchronizedcounter;

public class SynchronizedCounter {

        private long value;
```

```
        /**
         * Constructor with the initial value of the counter
         *
         * @param startValue
         *            the initial value
         */
        public SynchronizedCounter(long startValue) {
                this.value = startValue;
        }

        /**
         * Increment the counter
         */
        public synchronized void increment() {
                value++;
        }

        /**
         * Decrement the counter
         */
        public synchronized void decrement() {
                value--;
        }

        /**
         * Get the current value of the counter
         *
         * @return the cur rent counter value
         */
        public synchronized long getValue() {
                return value;
        }

        /**
         * Get the string representation of this Counter
         *
         * @return a String representing this counter
         */
        @Override
        public String toString() {
                return "" + value;
        }
}
```

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that objects variables are done through synchronized methods. This strategy is effective, but can introduce problems with liveness. Also this only provides a coarse-grained level to introduce synchronization. To have more fine-grained control one can use synchronized statements.

3.2 Synchronized statements

Another way to create synchronized code is with synchronized statements. Unlike synchronized methods, synchronized statements must specify the object to take the lock on.

```
Object lock = new Object();
synchronized(lock){
//synchronized actions
}
```

Each object can be used as lock, even the this object. One can also specify different locks for accessing different independent variables to allow fine-grained locking:

```
public class TwoCounters {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1(){
        synchronized (lock1){
            c1++;
        }
    }

    public void inc2(){
        synchronized (lock2){
            c2++;
        }
    }
}
```

Use this idiom with extreme care. Fine-grained locking can improve performance, but also enlarges the chance for deadlocks when not used properly.