

## Seminar 7 - Lecture

### Multithreading

### Laureate Online Education

When we read newspapers and we may want to listen to the radio at the same time. When I am working on this lecture note, I am listening to music on the same computer. I need my computer software to deal with this task – do more than one things at the same time. This is called the concurrent situation in computer software world. Java can deal with concurrent situation. There are both basic and high-level APIs in the Java. In this week, we will focus more on the higher level APIs in the `java.util.concurrent` packages.

### Overview of threads and processes

If you have taken the Operating System module the terms of Processes and Threads should be already familiar to you. If not, that is not problem, you will study them in this week. We will only address them from the viewpoint of the Java system. We will be studying two basic units of concurrent execution: **processes** and **threads**. In Java, thread control is mainly used for concurrent programming. Any computer system normally has many active processes and threads. It's becoming very common for a computer system to have multiple processors or processors with multiple execution cores. This increases a system's capacity to handle concurrent execution of processes and threads. It's worth pointing out that it is also possible to have concurrency on simple systems with single processor. Processing time for a single processor is shared among processes and threads through an OS feature called time slicing.

A **process** has a relatively independent and self-contained execution environment. It normally has a complete, private set of basic run-time resources and 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. IPC

is used not just for communication between processes on the same system, but processes on different systems.

Both threads and processes provide an environment. The big difference is creating a new thread requires fewer resources than creating a new process. Threads exist within a process — every process has at least one thread. Threads share the process's resources, including memory and open files. This is made for efficiency, but this is also potentially problematic. See the following figure quoted from Figure 23.1 in the textbook. It demonstrates different states of a thread, they are new, waiting (while it waits for another thread to perform a task), timed waiting (wait for a specific interval time), terminated (when it completed its task or otherwise terminate) and blocked (when it attempted to perform a task that cannot be completed immediately).

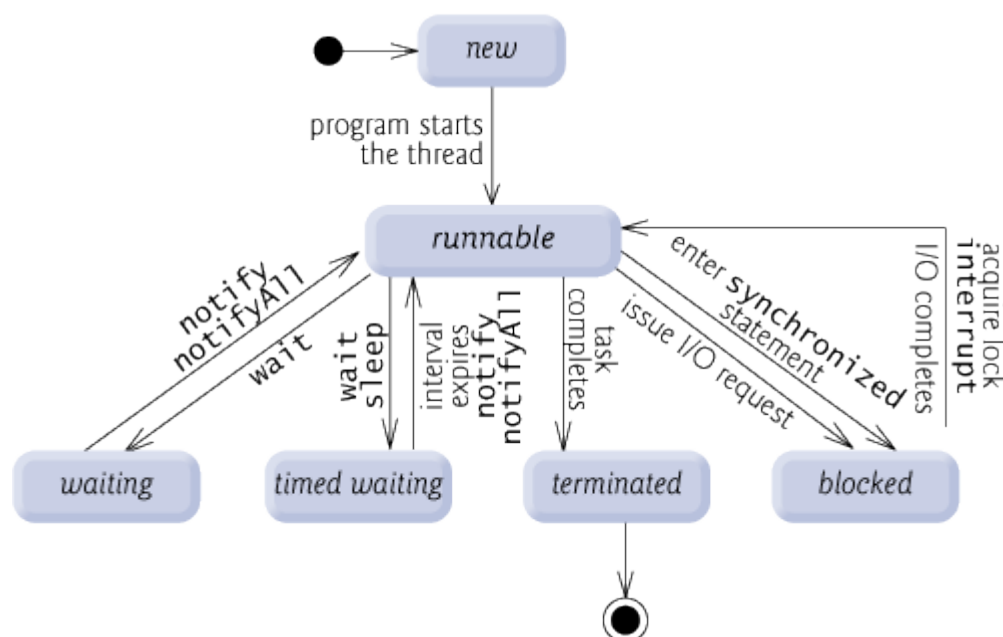


Figure 7-1, quoted from the textbook Fig 23.1 Thread life-cycle UML state diagram.

Each application should have at least one thread. As an application programmer, you should start with just one thread - the main thread. This main thread then has the ability to create further threads. Each thread is associated with an instance of the class Thread. There are two basic approaches for using Thread objects to create a concurrent application.

- To instantiate Thread each time the application needs to initiate an asynchronous task.
- To pass the application's tasks to an executor.

An application that creates a Thread instance must provide the code that will run in that thread. There are two ways to do this:

The first option is to 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:

```
1 public class HelloRunnable implements Runnable {  
2  
3     public void run() {  
4         System.out.println("Hello from a thread!");  
5     }  
6  
7     public static void main(String args[]) {  
8         (new Thread(new HelloRunnable())).start();  
9     }  
10 }  
11
```

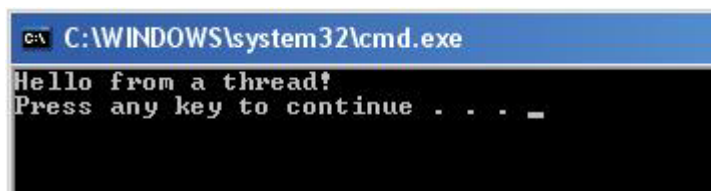


Figure 7-2 The HelloRunnable example and the output.

The second option is to design a subclass of 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:

```
1 public class HelloThread extends Thread
2 {
3     public void run()
4     {
5         System.out.println("Hello from a thread!");
6     }
7     public static void main(String args[])
8     {
9         (new HelloThread()).start();
10    }
11 }
12
```

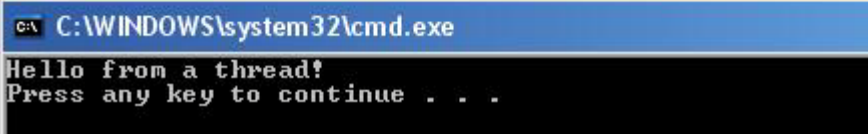


Figure 7-3 The HelloThread example and the output.

The first approach, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread.

The second approach is easier to use in simple applications. Your task class must be a descendant of Thread. We will focus more on the first approach, which separates the Runnable task from the Thread object that executes the task.

The Thread class defines a number of methods useful for thread management. These include static methods that can be used to provide information about, affect the status of thread and thread invoking method, etc. The other methods are invoked from other threads involved in managing the thread and Thread object. We'll examine some of these methods in the following sections.

There are two important methods defined in Thread – sleep and interrupt.

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient approach to make processor time available to the other threads.

An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide

exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

The example in Fig 7-4 uses sleep to print messages at four-second intervals:

```
1 public class SleepMessages {
2     public static void main(String args[]) throws InterruptedException {
3         String importantInfo[] = {
4             "Mares eat oats",
5             "Does eat oats",
6             "Little lambs eat ivy",
7             "A kid will eat ivy too"
8         };
9
10        for (int i = 0; i < importantInfo.length; i++) {
11            //Pause for 4 seconds
12            Thread.sleep(4000);
13            //Print a message
14            System.out.println(importantInfo[i]);
15        }
16    }
17 }
18
```




Figure 7-4 The SleepMessages example and the output.

The InterruptedException in the main method in Fig7-4 is an exception that will throw while sleeping when another thread interrupts the current thread.

An **interrupt** is to inform a thread that it should stop what it is doing and do something else. The programmer will decide how a thread responds to an interrupt. The easy and common solution is for the thread to terminate. A thread invokes interrupt on the Thread object to be interrupted and it is important for the interrupted thread must support its own interruption. So how does it work? The thread simply returns from the run method after it catches that exception. See the following modified code to support interrupts:

```
int i = 0;
while (i < importantInfo.length) {
    //Pause for 4 seconds
    try {
        Thread.sleep(4000);
```

```

    } catch (InterruptedException e) {
        //We've been interrupted: no more messages.
        return;
    }
    //Print a message
    System.out.println(importantInfo[i]);
    i++;
}

```

If a thread goes on for a while without invoking a method that throws `InterruptedException`, It must periodically invoke `Thread.interrupted`. See the following code:

```

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

```

Interrupt status is used as a flag. Invoking `Thread.interrupt` sets interrupt status. If `Thread.interrupted` is invoked, interrupt status is cleared. `Thread.isInterrupted` can be used by a thread to query the interrupt status of another thread. When a method exits by throwing an `InterruptedException`, it clears interrupt status.

It is normal for a thread to use the `join` method to wait for the completion of another thread. `join()` method causes the current thread to pause execution until it terminates. Like `sleep`, `join` exits with an `InterruptedException` by an interrupt.

See example in Fig 7-5, there are two threads in SimpleThreads. The first is the main thread. The main thread creates a new thread `MessageLoop`. It will wait until it finishes. If the `MessageLoop` thread takes too long to finish, it will be interrupted.

```

1 public class SimpleThreads {
2     //Display a message, preceded by the name of the current thread
3     static void threadMessage(String message) {
4         String threadName = Thread.currentThread().getName();
5         System.out.format("%s: %s\n", threadName, message);
6     }
7     private static class MessageLoop implements Runnable {
8         public void run() {
9             String importantInfo[] = {
10                 "Mares eat oats",
11                 "Does eat oats",
12                 "Little lambs eat ivy",
13                 "A kid will eat ivy too"
14             };
15             try {
16                 for (int i = 0; i < importantInfo.length; i++) {
17                     //Pause for 4 seconds
18                     Thread.sleep(4000);
19                     //Print a message
20                     threadMessage(importantInfo[i]);
21                 }
22             } catch (InterruptedException e) {
23                 threadMessage("I wasn't done!");
24             }
25         }
26     }
27     public static void main(String args[]) throws InterruptedException {
28         //Delay, in milliseconds before we interrupt MessageLoop
29         //thread (default one hour).
30         long patience = 1000 * 60 * 60;
31         //If command line argument present, gives patience in seconds.
32         if (args.length > 0) {
33             try {
34                 patience = Long.parseLong(args[0]) * 1000;
35             } catch (NumberFormatException e) {
36                 System.err.println("Argument must be an integer.");
37                 System.exit(1);
38             }
39         }
40         threadMessage("Starting MessageLoop thread");
41         long startTime = System.currentTimeMillis();
42         Thread t = new Thread(new MessageLoop());
43         t.start();
44         threadMessage("Waiting for MessageLoop thread to finish");
45         //loop until MessageLoop thread exits
46         while (t.isAlive()) {
47             threadMessage("Still waiting...");
48             //Wait maximum of 1 second for MessageLoop thread to
49             //finish.
50             t.join(1000);
51             if (((System.currentTimeMillis() - startTime) > patience) &&
52                 t.isAlive()) {
53                 threadMessage("Tired of waiting!");
54                 t.interrupt();
55                 //Shouldn't be long now -- wait indefinitely
56                 t.join();
57             }
58         }
59         threadMessage("Finally!");
60     }
61 }
62

```



```
C:\WINDOWS\system32\cmd.exe
main: Starting MessageLoop thread
main: Waiting for MessageLoop thread to finish
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Mares eat oats
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Does eat oats
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Little lambs eat ivy
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: A kid will eat ivy too
main: Finally!
Press any key to continue . . . _
```

Figure 7-5 A SimpleThreads example and the output.

## Thread Synchronization

It is important for a Thread to access to shared fields and the objects in order to communicate with other Threads. This form of communication is very efficient, but it is also possible to create new problems: thread interference and memory consistency errors. The tool needed to prevent these errors is synchronization.

Intrinsic locks are used to implement synchronization. There is an intrinsic lock associated with each object in Java. A thread has to acquire the object's intrinsic lock before accessing them in order to have exclusive and consistent access to an object's fields. The intrinsic lock should be released when it's done with them. A thread owns an intrinsic lock exclusively between the time it has acquired the lock and released the lock.

In order to start a synchronized method, a thread will automatically acquire the intrinsic lock for that method's object and release it on the method's return. It will happen even if the return was caused by exceptions.



## Synchronized Statements

One way to create synchronized code is to use synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
synchronized(object)
{
    statements
} // end synchronized statement
```

In addition to synchronized statement, we can also use synchronized methods. To define a method as synchronized, just place the synchronized keyword before the method's return type.

Compare to the following two cases: **unsynchronized data sharing** and **synchronized data sharing**

### Unsynchronized data sharing

See this singleArray example at Fig 23.7, 23.8 and 23.9 in the textbook. Two Runnables maintain references to a single integer array. The output in Fig. 23.9 in the textbook demonstrates the problems (highlighted in the output) that can be caused by failure to synchronize access to shared data. The value 1 was written to element 0, then overwritten later by the value 11. Also, when writeIndex was incremented to 3, nothing was written to that element, as indicated by the 0 in that element of the printed array. To prevent this, we need to use **synchronized data sharing**, see textbook Fig 23.10 in the textbook.

An atomic action is an operation that cannot be divided into smaller sub-operations. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete. There are actions you can specify that are atomic: **Reads and writes** are atomic for reference variables and for most primitive variables.

## Multithread in GUI

In a Swing application, each Swing components are accessed from only one single thread, called the event dispatch thread. This is to ensure the safety in GUI applications, because all tasks can be managed and monitored in one place. All tasks that require interaction with an application's GUI are placed in an event queue and are executed sequentially by the event dispatch thread.

It is fine to perform simple calculations on the event dispatch thread in sequence with GUI component manipulations. This may cause problems when an application must perform a lengthy computation in response to a user interface interaction. This is because the event dispatch thread cannot attend to other tasks in the event queue while the thread is busy doing that computation. This causes the GUI components to become unresponsive. The solution is to have more than one thread. We can handle the long-running computation in a separate thread, this can free the event dispatch thread to continue managing other GUI interactions. Please refer back to the textbook chapter 23.11 for examples.

## **Required Reading**

In Chapter 23 of the text, read the following sections:

23.1 - 23.5  
Skim 23.6-23.8

## **Helpful Additional Readings**

## **Helpful Websites**

<http://java.sun.com/docs/books/tutorial/essential/concurrency>

## **Acknowledgements**

The examples in Fig 7-2, 7-3, 7-4 and 7-5 for this lecture were drawn from the above web site and according to SUN's directive

are copyright protected according to the following copyright message:

```
/*
 * Copyright (c) 1995 - 2008 Sun Microsystems, Inc. All rights
reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * - Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * - Redistributions in binary form must reproduce the above
copyright
 * notice, this list of conditions and the following disclaimer
in the
 * documentation and/or other materials provided with the
distribution.
 *
 * - Neither the name of Sun Microsystems nor the names of its
 * contributors may be used to endorse or promote products
derived
 * from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS
 * IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```