

Threads

61FIT3NPR -Network Programming

Faculty of Information Technology
Hanoi University
Fall 2020



Lesson's Objectives

- By the end of this lesson you will:
 - Be familiar with the Java threads syntax and API
 - Be able to create Java multithreaded applications

Agenda

- Threads Overview

- Creating threads in Java

- Synchronization

- wait() and notify()

- Thread Pools

- Exercise

Threads Overview

- Threads allow the program to **run tasks in parallel**
- In many cases threads need to be **synchronized**, that is, be kept not to handle the same data in **memory** concurrently
- There are cases in which a thread needs to **wait** for another thread before proceeding

Never use thread-per-session – this is a wrong and un-scaled architecture – use instead Thread Pools

Agenda

Threads Overview

- Creating threads in Java

Synchronization

wait() and notify()

Thread Pools

Exercise

Threads in Java

The operation we want to be threaded:

```
public class PrintNumbers {  
    static public void printNumbers() {  
        for(int i=0; i<1000; i++) {  
            System.out.println(  
                Thread.currentThread().getId() +  
                    ": " + i);  
        }  
    }  
}
```

Threads in Java

Option 1 – extending class Thread:

```
public class Thread1 extends Thread {  
  
    @Override  
    public void run() {  
        System.out.println("Thread1 ThreadId: " +  
            Thread.currentThread().getId());  
        // do our thing  
        PrintNumbers.printNumbers();  
        // the super doesn't anything,  
        // but just for the courtesy and good practice  
        super.run();  
    }  
}
```

Threads in Java

Option 1 – extending class Thread (cont’):

```
static public void main(String[] args) {  
    System.out.println("Main ThreadId: " +  
        Thread.currentThread().getId());  
    for(int i=0; i<3; i++) {  
        new Thread1().start(); // don't call run!  
        // (if you want a separate thread)  
    }  
    printNumbers();  
}
```


Threads in Java

Option 2 – implementing Runnable:

```
public class Thread2 implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println("Thread2 ThreadId: " +  
            Thread.currentThread().getId());  
        // do our thing  
        PrintNumbers.printNumbers();  
    }  
}
```

Threads in Java

Option 2 – implementing Runnable (cont'):

```
static public void main(String[] args) {  
    System.out.println("Main ThreadId: " +  
        Thread.currentThread().getId());  
    for(int i=0; i<3; i++) {  
        new Thread(new Thread2()).start();  
        // again, don't call run!  
        // (if you want a separate thread)  
    }  
    printNumbers();  
}
```

Threads in Java

Option 3 – implementing Runnable as Anonymous:

```
static public void main(String[] args) {
    System.out.println("Main ThreadId: " +
        Thread.currentThread().getId());
    new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("Thread3 ThreadId: " +
                Thread.currentThread().getId());
            // do our thing
            printNumbers();
        }
    }).start(); // don't call run! ...
    printNumbers();
}
```

Agenda

Threads Overview

Creating threads in Java

- Synchronization

wait() and notify()

Thread Pools

Exercise

Synchronization

Synchronization of threads is needed for in order to control threads coordination, mainly in order to **prevent simultaneous operations on data**

For simple synchronization Java provides the **synchronized** keyword

For more sophisticated **locking mechanisms**, starting from Java 5, the package `java.concurrent.locks` provides additional locking options, see:

<http://java.sun.com/javase/6/docs/api/java/util/concurrent/locks/package-summary.html>

Synchronization

Example 1 – synchronizing methods:

```
public class SynchronizedCounter {  
    private int c = 0;  
    public synchronized void increment() { c++; }  
    public synchronized void decrement() { c--; }  
    public synchronized int value() { return c; }  
}
```

The **synchronized** keyword on a method means that if this is already locked anywhere (on this method or elsewhere) by another thread, we need to wait till this is unlocked before entering the method

Reentrant
is allowed

Synchronization

Example 2 – synchronizing blocks:

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

When **synchronizing a block**, key for the locking should be supplied (usually would be **this**)
The advantage of not synchronizing the entire method is **efficiency**

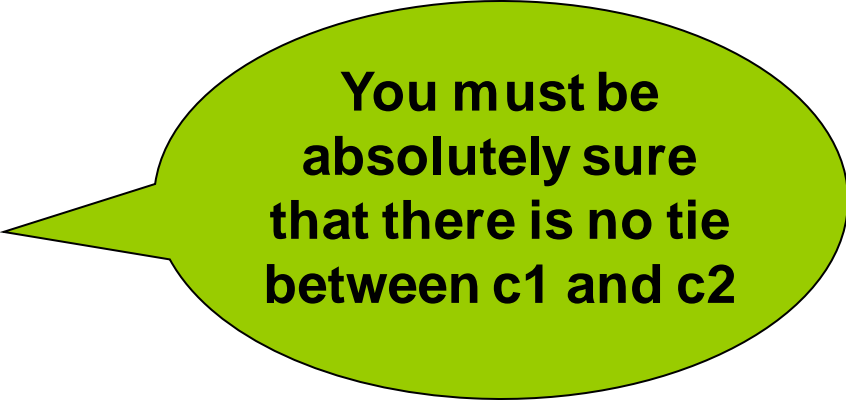
Synchronization

Example 3 – synchronizing using different locks:

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

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

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



**You must be
absolutely sure
that there is no tie
between c1 and c2**

```
}
```


Synchronization

Example 4 – synchronizing static methods:

```
public class Screen {  
    private static Screen theScreen;  
    private Screen(){...} // private c'tor  
    public static synchronized getScreen() {  
        if(theScreen == null) {  
            theScreen = new Screen();  
        }  
        return theScreen;  
    }  
}
```

**This is a
Singleton
example**

**It is not the most
efficient way to implement
Singleton in Java**

Synchronization

Example 4 – synchronizing static methods ...

Having a **static method** be **synchronized** means that **ALL objects of this type are locked** on the method and can get in one thread at a time.

The lock is the **Class object** representing this class.

The performance penalty might be sometimes too high – **needs careful attention!**

Synchronization

Example 4' – a better singleton:

```
public class Screen {  
    private static Screen theScreen = new Screen();  
    private Screen(){...} // private c'tor  
    public static getScreen() {  
        return theScreen;  
    }  
}
```



**No
synchronization**

Agenda

Threads Overview

Creating threads in Java

Synchronization

- wait() and notify()

Thread Pools

Exercise

wait(), notify(), notifyAll()

This is an optional topic

We may skip it...

wait(), notify(), notifyAll()

wait() and **notify()** allows a thread to **wait for an event**

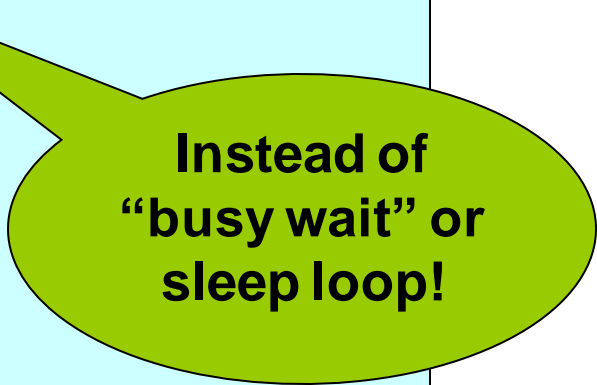
A call to **notifyAll()** allows all threads that are on **wait()** with the same lock to be released

A call to **notify()** allows one arbitrary thread that is on a **wait()** with the same lock to be released

Read:

(a) <http://java.sun.com/docs/books/tutorial/essential/concurrency/guardmeth.html>

(b) [http://java.sun.com/javase/6/docs/api/java/lang/Object.html#wait\(\)](http://java.sun.com/javase/6/docs/api/java/lang/Object.html#wait())



Instead of
“busy wait” or
sleep loop!

wait(), notify(), notifyAll()

Example

(from <http://java.sun.com/docs/books/tutorial/essential/concurrency/example/Drop.java>):

```
public class Drop {
```

```
    // Message sent from producer to consumer  
    private String message;
```

```
    // A flag, True if consumer should wait for  
    // producer to send message, False if producer  
    // should wait for consumer to retrieve message  
    private boolean empty = true;
```

```
    ...
```

**Flag must be used, never
count only on the notify**

wait(), notify(), notifyAll()

Example (cont')

```
public class Drop {  
    ...
```

```
    public synchronized String take() {
```

```
        // Wait until message is available
```

```
        while (empty) {
```

```
            // we do nothing on InterruptedException
```

```
            // since the while condition is checked anyhow
```

```
            try { wait(); } catch (InterruptedException e) {}
```

```
        }
```

```
        // Toggle status and notify on the status change
```

```
        empty = true;
```

```
        notifyAll();
```

```
        return message;
```

```
    }
```

```
    ...
```

```
}
```

**Must be in
synchronized context**

wait(), notify(), notifyAll()

Example (cont')

```
public class Drop {  
    ...
```

```
    public synchronized void put(String message) {
```

```
        // Wait until message has been retrieved
```

```
        while (!empty) {
```

```
            // we do nothing on InterruptedException
```

```
            // since the while condition is checked anyhow
```

```
            try { wait(); } catch (InterruptedException e) {}
```

```
        }
```

```
        // Toggle status, store message and notify consumer
```

```
        empty = false;
```

```
        this.message = message;
```

```
        notifyAll();
```

```
    }
```

```
    ...
```

```
}
```

**Must be in
synchronized context**

Agenda

Threads Overview

Creating threads in Java

Synchronization

wait() and notify()

■ Thread Pools

Exercise

Thread Pools

Prevernt the thread-per-session pitfall!

Class ThreadPoolExecutor:

<http://java.sun.com/javase/6/docs/api/java/util/concurrent/ThreadPoolExecutor.html>

Agenda

Threads Overview

Creating threads in Java

Synchronization

wait() and notify()

Thread Pools

■ Exercise

Exercise

Implement a multithreaded application performing **X sessions** of the `PrintNumbers.printNumbers` task, (presented at the beginning of this lesson) – with a Thread Pool of **Y threads**

X and Y should be retrieved from the command-line

Thread Example

```
package tut4;

public class HelloMain {
    public static void main(String[] args) throws InterruptedException {
        int idx = 1;
        for (int i = 0; i < 2; i++) {
            System.out.println("Main thread running " + idx++);
            // Sleep 2101 milliseconds.
            Thread.sleep(2101);
        }
        HelloThread helloThread = new HelloThread();
        // Run thread
        helloThread.start();
        for (int i = 0; i < 3; i++) {
            System.out.println("Main thread running " + idx++);
            // Sleep 2101 milliseconds.
            Thread.sleep(2101);
        }
        System.out.println("==> Main thread stopped");
    }
}
```

Thread Example

```
package tut4;

public class HelloThread extends Thread {
    @Override
    public void run() {
        int index = 1;

        for (int i = 0; i < 10; i++) {
            System.out.println("  - HelloThread running " + index++);

            try {
                // Sleep 1030 milliseconds.
                Thread.sleep(1030);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("  - ==> HelloThread stopped");
    }
}
```



Thank You