COMPSCI 230 S2C 2013
Software Design and Construction

Introduction to Java Threads
Lecture 1 of Theme C

Lecture Plan for Weeks 10-12

18/5	Introduction to Java threads	Sikora pp. 157-9, Goetz1 pp. 1-6.
21/5	A thread's life	Goetz1 pp. 6-10.
22/5	Where Java threads are used; synchronization	Goetz1 pp. 10-15.
25/5	Locking, blocking, mutex; visibility, consistency.	Goetz1 pp. 15-20.
28/5	Deadlock; performance; programming guidelines.	Goetz1 pp. 20-24.
29/5	Dealing with InterruptedException (intro)	Goetz2 pp. 1-3.
1/6	Executors, tasks, concurrent collections, synchronizers.	Bloch pp. 271-7.
4/6	Concurrency in Swing	Oracle
5/6	Debugging Swing / Revision of this unit	Potochkin

Readings for this unit

- Strongly recommended!

1. **Sikora**
   - Zbigniew Sikora, "Threads", Chapter 10 of *Java: Practical Guide for Programmers*, Elsevier, 2003. Available to registered students through our library: http://www.sciencedirect.com.ezproxy.auckland.ac.nz/science/article/pii/B9781558609099500107
2. **Goetz1**
   - Brian Goetz, "Introduction to Java threads", IBM developerWorks, 26 Sep 2002, 27 pages. Available: http://www.ibm.com/developerworks/java/tutorials/j-threads/j-threads-pdf.pdf
3. **Goetz2**
   - Brian Goetz, "Java theory and Practice: Dealing with InterruptedException", IBM developerWorks, 23 May 2006. Available: http://www.ibm.com/developerworks/java/library/j-jtp05236/index.html
4. **Bloch**
   - Joshua Bloch, "Concurrency: Prefer executors and tasks to threads, and Prefer concurrency utilities to wait and notify", Items 68 and 69 in Chapter 10 of *Effective Java*, Prentice Hall, 2nd Edition, 2008. Available to registered students through our library: http://proquestcombo.safaribooksonline.com.ezproxy.auckland.ac.nz/9780137150021
5. **Oracle**
   - Oracle, "Lesson: Concurrency in Swing", The Java Tutorials, 2013. Available: http://docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html
6. **Potochkin**
   - Alexander Potochkin, "Debugging Swing, the final summary", 16 February 2006. Available: https://weblogs.java.net/blog/alexfromsun/archive/2006/02/debugging_swing.html

Learning Goals for Today

- Develop an appropriate "mental model" for multithreaded programs.
  - Predict the outputs of a simple multithreaded program.
- Understand why multithreading is important – and difficult!
  - List, and briefly discuss, some of the ways in which multithreading is used in Java.
  - Recognise some common "design patterns" for multithreaded computations: Model-View-Controller, simulation with one-thread-per-actor, foreground/background computations.
  - Explain how a `volatile` variable differs from a non-volatile one: what are its advantages and disadvantages?

PrintNumbersThread

```
public class PrintNumbersThread extends Thread {
  String name;
  public PrintNumbersThread( String threadName ) {
    name = threadName;
  }
  public void run() {
    for( int i=1; i<=2; i++ ) {
      System.out.println(name + ": " + i);
      try { Thread.sleep(500); }
        catch( InterruptedException e ) { }
    }
  }
}
```

RunThreads

```
public class RunThreads {
  public static void main( String args[] ) {
    PrintNumbersThread threadl;
    PrintNumbersThread thread2;
    threadl = new PrintNumbersThread("Threadl");
    thread2 = new PrintNumbersThread("Thread2") ;
    threadl.start () ;
    thread2.start () ;
  }
}
```

Expected output:

```
Thread1: 1
Thread2: 1
Thread1: 2
Thread2: 2
```

or

```
Thread1: 1
Thread2: 1
Thread2: 2
Thread1: 2
```

```
Thread2: 1
Thread1: 1
Thread2: 2
Thread1: 2
```

```
Thread2: 1
Thread1: 2
Thread1: 1
Thread2: 2
```

```
Thread1: 1
Thread1: 2
Thread2: 1
Thread2: 2
```

```
Thread2: 1
Thread2: 2
Thread1: 1
Thread1: 2
```

Tracing a Threaded Program

```
public class RunThreads {
  public static void main( String args[] ) {
    PrintNumbersThread threadl;
    PrintNumbersThread thread2;
    threadl = new PrintNumbersThread("Threadl");
    thread2 = new PrintNumbersThread("Thread2") ;
    threadl.start () ;
    thread2.start () ;
} }
public class PrintNumbersThread extends Thread {
  public void run() {
    for( int i=1; i<3; i++ ) {
      System.out.println(name + ": " + i);
      try { Thread.sleep(500); }
        catch( InterruptedException e ) { }
  } }
```

```
Thread1: 1
Thread2: 1
Thread1: 2
Thread2: 2
```

Note that the “parent thread” dies before its children.

Tracing a Threaded Program (2)

```
public class RunThreads {
  public static void main( String args[] ) {
    PrintNumbersThread threadl;
    PrintNumbersThread thread2;
    threadl = new PrintNumbersThread("Threadl");
    thread2 = new PrintNumbersThread("Thread2") ;
    threadl.start () ;
    thread2.start () ;
} }
public class PrintNumbersThread extends Thread {
  public void run() {
    for( int i=1; i<3; i++ ) {
      System.out.println(name + ": " + i);
      try { Thread.sleep(500); }
        catch( InterruptedException e ) { }
  } }
```

```
Thread1: 1
Thread2: 1
Thread2: 2
Thread1: 2
```

Note that the threads are unsynchronised.

CPUs, Cores, Processes, Threads

- Modern computers have many forms of parallelism.
- In hardware, there are
  - One to four CPU chips, with
  - Two to eight cores per CPU chip, and
  - Hundreds of instructions in the execution pipeline of each core.
- In software, there are
  - Hundreds of processes, where
    - Each process is either running or waiting (for a core or an I/O device); and
  - One to 20 (or more) threads of control per process.
    - Each thread is either running or waiting.
    - (There are actually four states in Java's thread model, as we'll see later.)
- If you are "hand-executing" a multithreaded program, you probably move only one instruction-pointer at a time – this is like a single-core execution.
  - If you could move 8 pointers simultaneously, you'd be simulating an 8-core CPU.

Context Switches

- When a core "switches" its context to start executing a different thread, there is significant performance penalty:
  - Very roughly: hundreds of "wasted" instruction-execution cycles.
  - When you're hand-executing a multi-threaded program, you have to 'move your hand' to a different instruction-pointer before you can start to move it – this is your context-switching time.
- Currently, most CPU cores run only one thread at a time.
  - Ideally: number of runnable threads ≈ number of cores.
  - Ideally: each thread runs for a long time (>> 1000 instructions) before it has to "wait" for the output of another thread, or for an I/O device, or before it is interrupted by the end of its time-slice.
  - Currently, most operating systems have 100 to 1000 time-slices per second.
- If you have more than 100 threads in a single Java program on a laptop or home computer, your threads will be waiting most of the time.
  - If threads have to wait more than 30 msec, your GUI will probably be "jerky" and "sluggish".

Parallelism is difficult, why use it?

- "Because it's there"
  - If you write single-threaded Java programs, and your competitors are multi-threading efficiently, their programs will run 3x or even 8x faster because they're using CPU cores that you're leaving idle.
    - This is especially noticeable on "CPU-limited" computations e.g. image analysis.
    - Note: modern PCs also have a GPU (Graphics Processor Unit), allowing very efficient computer graphics without burdening the CPU.
- "Because it's very convenient"
  - When you're writing GUIs, you generally use one thread to render the graphics (the "View"), one or more threads to run the back-end computation (the "Model"), and one thread (the "Controller") to accept input from the user.
  - If you single-thread a GUI, the controls will be non-responsive and the display will "freeze" while you're updating the Model (unless your model-updates take 30 milliseconds or less).

Why use parallelism? (cont.)

- "Because it's built into the JVM"
  - The JVM has some daemon threads which run very helpful services, e.g. its memory "garbage" collection.
    - In earlier languages, you had to "clean up your own garbage" by explicitly de-allocating objects.
    - Java collects "garbage" objects automatically – and correctly almost-all of the time, unless you terminate your threads improperly!
  - JVM's daemons are carefully designed to "stay out of your way":
    - running only when necessary,
    - making useful progress during a single time-slice, and
    - allowing your program's threads to make progress (on other CPU cores) while the service is actively running .

Why use parallelism? (cont.)

- “Because it’s natural (in some programs)”
  - When simulating a system with many actors, it’s natural to have one thread per actor:
    - the thread’s run() method describes “what this actor does”.
  - For example, a traffic simulator might have one thread for each automobile, bus, or truck that is on the roads being simulated.
  - Warning: using “parallelism to fit your problem,” rather than “parallelism to fit your hardware”, may lead to very inefficient computations.
    - A desktop PC will not run 10000 threads efficiently, however it can efficiently simulate 10000 automobiles in a roading network (if your simulator uses 100 threads).

Why use parallelism? (cont.)

- “Because it’s natural (in some programs)”
  - In a server program, a thread “worker” can be assigned to each client.
    - The thread’s run() method delivers the service.
  - In a program that handles asynchronous I/O devices (e.g. network interfaces, disks, keyboards) a thread can be assigned to each device.
    - The thread’s run() method handles the I/O stream for this device.
    - If the thread executes a blocking read, e.g. SocketInputStream.read(), it will not run again until the read succeeds. The JVM handles this wait very efficiently.

Sharing Nicely

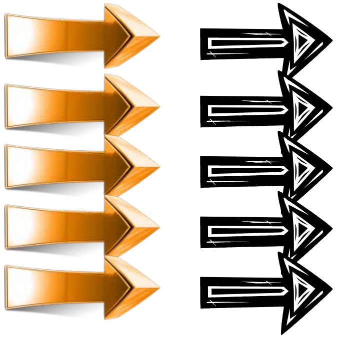
- If your threads don't "talk" to each other, they can't cooperate.
- If your threads do "talk", they might confuse each other.
  - When one thread is changing an object, the other threads must be prevented from reading this object until the changes are complete.
  - When one thread is accessing a method, other threads must wait their turn (unless the method is "thread-safe" i.e. it can handle multiple simultaneous accesses).
- There are several ways to share safely…

Volatile variables

- If a variable, object, or field is declared as volatile, then
  - It can be used for reliable communication between threads.
- Non-volatile variables, objects, and fields have unpredictable semantics, if they are read & written by more than one thread.
  - For example, if Thread1 and Thread2 are both executing the following:

```
int x = 1;
System.out.println(name + ": " + ( x++ ) );
```

  - This is equivalent to executing:



```
int x = 1;
int t = x;
t = t + 1;
x = t;
System.out.println(name + ": " + x );
```

Thread1: 2
Thread2: 3

Volatile variables (2)

- If a variable, object, or field is declared as volatile, then
  - It can be used for reliable communication between threads.
- Non-volatile variables, objects, and fields have unpredictable semantics, if they are read & written by more than one thread.
  - For example, if Thread1 and Thread2 are both executing the following:

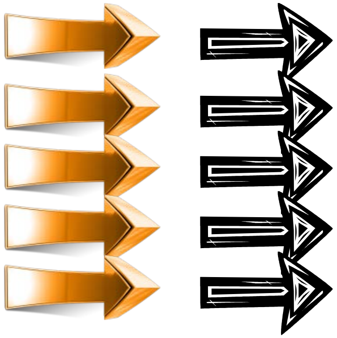
```
int x = 1;
System.out.println(name + ": " + ( x++ ) );
```

```
Thread1: 2
Thread2: 3
```

or

```
Thread2: 2
Thread1: 3
```

  - This is equivalent to executing:



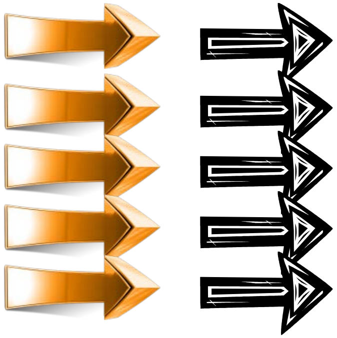
```
int x = 1;
int t = x;
t = t + 1;
x = t;
System.out.println(name + ": " + x );
```

Volatile variables (3)

- If a variable, object, or field is declared as volatile, then
  - It can be used for reliable communication between threads.
- Non-volatile variables, objects, and fields have unpredictable semantics, if they are read & written by more than one thread.
  - For example, if Thread1 and Thread2 are both executing the following:

```
int x = 1;
System.out.println(name + ": " + ( x++ ) );
```

  - This is equivalent to executing:



```
int x = 1;
int t = x;
t = t + 1;
x = t;
System.out.println(name + ": " + x );
```

```
Thread1: 2
Thread2: 3
```

or

```
Thread2: 2
Thread1: 3
```

or

```
Thread2: 3
Thread1: 2
```

or

```
Thread1: 3
Thread2: 2
```

or

```
Thread1: 2
Thread2: 2
```

or

```
Thread2: 2
Thread1: 2
```

Volatile variables (4)

- If a variable, object, or field is declared as volatile, then
  - It can be used for reliable communication between threads.
  - Semantics are predictable – if a thread reads the variable then writes it, the other thread is blocked from reading until the newly-written value is available.
    - Warning: you can cripple a multithreaded program by making all of its variables volatile.
    - The JVM must always read volatiles from memory. Frequently-used non-volatile values are retained in the CPU register file, which is *much* faster than main memory.
  - For example, if Thread1 and Thread2 are both executing the following:

```
volatile int x = 1;
System.out.println(name + ": " + ( x++ ) );
```

  - Thread1 and Thread2 always get different values!

```
Thread1: 2
Thread2: 3
```

or

```
Thread2: 2
Thread1: 3
```

```
Thread1: 3
Thread2: 2
```

or

```
Thread2: 3
Thread1: 2
```

Learning Goals for Today

- Develop an appropriate "mental model" for multithreaded programs.
  - Predict the outputs of a simple multithreaded program.
- Understand why multithreading is important – and difficult!
  - List, and briefly discuss, some of the ways in which multithreading is used in Java.
  - Recognise some common "design patterns" for multithreaded computations: Model-View-Controller, simulation with one-thread-per-actor, foreground/background computations.
  - Explain how a `volatile` variable differs from a non-volatile one: what are its advantages and disadvantages?