COMPSCI 230 S2C 2015
Software Design and Construction

Thread usage and synchronisation
Lecture 3 of Theme C

Lecture Plan for Weeks 7-9

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

Learning Goals for Today

- Distinguish daemons from user threads
  - How are they different? What are they doing in your JVM?
- What are some of the common uses of multithreading in Java?
  - What is the "thread architecture" of AWT/Swing? Which tasks belong on which thread? What can happen if the EDT is handling tasks that belong on the model or controller thread?
  - What is a TimerTask, an RMI, a servlet, and a JSP? When might I want to use these libraries in Java?
- Develop a working understanding of synchronization
  - What are locks? Atomic operations? Synchronized methods? When should I use them?
  - What can happen if an application has defective synchronization?

Daemon Threads

- "We mentioned that a Java program exits when all of its threads have completed, but this is not exactly correct. …" (Goetz 2002)
  - The JVM has some threads which only terminate when the JVM is terminated – these are called daemons.
    - The JVM's garbage collector is a daemon which does a good (but not perfect) job of "cleaning the sandbox" – by reclaiming memory that is consumed by objects that are no longer needed.
    - Subsequent applications or servlets need memory for their objects. Garbage collection is a very important service!
- "The Java Virtual Machine exits when the only threads running are all daemon threads." (Java SE7, SE6, …)
  - Any thread can call `Thread.setDaemon()`. It then becomes a daemon!
  - The garbage collector is a daemon thread, created by the JVM.
  - User-created daemons are necessary if you want to configure a JVM as a server.
    - You'll want a daemon to handle your console input.
    - Another daemon handles service requests, e.g. http://commons.apache.org/daemon/.
  - User-created daemons are dangerous, from a security perspective.
    - Subsequent applications or servlets don't always get a "clean sandbox"!

Servlets

- Recent editions of Java EE have included the Servlet class. History:
  - 1999: Servlet 2.1 is part of J2EE 1.2 (the "Enterprise Edition" of Java)
  - 2005: Servlet 2.5 is released for J2EE 1.4.
  - 2009: Servlet 3.0 released for Java EE 6.
  - 2013: Servlet 3.1 released for Java EE 7.
- We won't study Servlets carefully – this is an advanced topic. Roughly…
  - A servlet is similar to a Java applet, but it is running on a remote JVM that is configured to be a server.
  - It's commonly used with HTTP, in the `javax.servlet.http` package.
    1. A user's browser issues an `HTTP get` request
    2. This request is handled by a "Servlet container" on the webserver.
    3. The container (on a user thread in the server's JVM) invokes the `init()` method of the appropriate servlet (if the servlet is not running already).
    4. A running servlet invokes a `service()` method, which spawns a thread to handle this user's request and any future requests from this user (during this session).

Java Server Pages (JSP)

- This was Sun Microsystem's response to the Active Server Pages (ASP) technology of Microsoft.
  - ASP was an optional part of the Internet Information Services (IIS) for Windows NT 4.0, in 1998 (?).
    - IIS is a web server, mail server, and FTP server.
      - IIS was Microsoft's answer, in1993 (?), to the NCSA HTTPd codebase… which Berners-Lee developed in 1990, and which morphed into Apache.
- Usually, JSP provides the "View" of a web-server's MVC architecture, in which JavaBeans is the "Model" and Servlets (or some other framework) is the "Controller".
  - I will not discuss JSP in any more detail, but I'd suggest you start with http://en.wikipedia.org/wiki/JavaServer_Pages if you want to learn more.

The Model-View-Controller Design Pattern

- You have seen the MVC pattern already, in Swing/AWT.
  - Note: the Model and View are not always clearly distinguished in a Swing app.
- Goetz' explanation of AWT:
  - "The AWT toolkit creates a single thread for handling UI events, and any event listeners called by AWT events execute in the AWT event thread."
    - This thread is commonly called the EDT, or "Event dispatching thread".
    - It is very important to run only short, non-blocking tasks on this thread.
    - A Java GUI will "feel" very unresponsive if its EDT is running tasks which take more than 30 milliseconds to run.
    - If an EDT runs a task that takes seconds to complete, the GUI will be "frozen" during this period.
  - "… you have to find a way for long-running tasks triggered by event listeners – such as checking spelling in a large document or searching a file system for a file – to run in a background thread so the UI doesn't freeze while the task is running…"
    - "A good example of a framework for doing this is the SwingWorker class."

TimerTasks

- The TimerTask framework is a convenient way to run tasks on a periodic schedule.
  - A thread can put itself to sleep, but it is more elegant (= more maintainable) to factor the scheduling code from the task-specfic code.
  - The TimerTask handles the scheduling – it can run a task every 100 msec, every 2000 msec, or at any other rate (which can be adjusted at runtime).
  - The syntax is straightforward, as seen on the next slide.

Goetz1, p. 11: TimerTask example

```
public static void main( String[] args ) {
  Timer timer = new Timer();

  final CalculatePrimes calculator = new CalculatePrimes();
  calculator.start();
  timer.schedule(
    new TimerTask() {
      public void run() {
        calculator.finished = true;
      }
    },
    TEN_SECONDS
  );
}
```

Threads can Work Cooperatively!

- The simplest communication mechanism is a shared variable.
  - Threads must be very careful to avoid writing to the same variable at the same time.
    - If two threads write simultaneously, at most one of these writes will succeed.
    - In the worst case, both writes succeed partially (in different portions of a shared object), and the object has a corrupted/inconsistent value.
  - To avoid concurrent writing on an object, you can use a boolean (or other single-word primitive) variable. You'll need a protocol, for example:
    - The "master" thread sets `flag=true` when it is safe for the "slave" thread to write to the object.
    - The "slave" resets the flag (`flag=false`) after it has written to the object.
    - The "master" can write to the object safely when (`flag == false`) .
    - Warning: the flag must be `volatile`, otherwise the slave may never see a `true` value.
      - In modern computer systems, thousands (or millions) of memory locations are cached by each CPU chip. Each core may have a separate cache.
      - A write to "memory" may not be visible to another core for a long time…

Synchronized variables

- If you have more than a few boolean flags in your code, or a complicated protocol for sharing, you'll probably have bugs.
  - It'll certainly be difficult to gain confidence that your code is bug-free!
  - Timing bugs can be very difficult to track down – they tend to be intermittent (i.e. not reliably exposed by a simple JUnit test), depending on difficult-to-control factors such as the CPU and memory workload of other processes on the system that is running your JVM.
- Synchronized objects are a convenient way to ensure
  - Atomicity: No more than one thread is writing to the object at any given time.
    - Each write operation is completed (on all fields of the object) before any other write operation is allowed to start.
  - Visibility: The writes of one thread are exposed to other threads, the next time they read the object.
    - The `volatile` keyword assures visibility, but it does not assure atomicity.

Monitors and Locks

- Java synchronization is based on an underlying technology (supported by every modern operating system and CPU) called "locks".
  - A lock is a volatile boolean variable with a very cleverly-designed protocol.
    - I will not discuss locking protocols in COMPSCI 230 – this is an advanced topic in parallel computing!
  - Any thread can "acquire a lock" if it asks for it… and if it is willing to wait… perhaps for a very long time… (perhaps forever! – this program defect is called "deadlock")
    - It is very important for every thread to "release a lock" as soon as possible, otherwise other threads may be waiting a long time.
  - Every Java object has a lock – making it somewhat thread-safe (because only one thread can change it at a time). We'll discuss thread-safety later...
  - If you declare a block of code to be `synchronized`, it becomes a "monitor" – meaning that only one thread can be executing it at any given time.

Learning Goals for Today

- Distinguish daemons from user threads
  - How are they different? What are they doing in your JVM?
- What are some of the common uses of multithreading in Java?
  - What is the "thread architecture" of AWT/Swing? Which tasks belong on which thread? What can happen if the EDT is handling tasks that belong on the model or controller thread?
  - What is a TimerTask, an RMI, a servlet, and a JSP? When might I want to use these libraries in Java?
- Develop a working understanding of synchronization
  - What are locks? Atomic operations? Synchronized methods? When should I use them?
  - What can happen if an application has defective synchronization?