COMPSCI 230 S2C 2013
Software Design and Construction

Deadlock, Performance, Programming Guidelines
Lecture 6 of Theme C

Learning Goals for Today

- Learn a little more Java:
  - `wait(), notify(), notifyAll().`
  - I do not expect you to be able to write code which invokes these methods appropriately.
  - The syntax is uncomplicated, but the code-design issues are *very* difficult.
- You *may* be examined on
  - Your understanding of the ways in which threads can safely signal each other, without "stepping on" each others' variables.
  - Your analysis of a multithreaded code, to determine whether or not there is some inappropriate interaction between its thread which may lead to deadlock or to corrupted computations.

wait(), notify(), and notifyAll()

- Goetz: "In addition to using polling,
  - which can consume substantial CPU resources and has imprecise timing characteristics,
  - the Object class includes several methods for threads to signal events from one thread to another."

- Note: Goetz used polling in his TimerTask example.
  - Let's review that example now.
- Polling is a very important design pattern! It is appropriate
  - whenever event-signalling isn't feasible, or
  - when the resource and time costs of polling are affordable, for example when the polling loop won't run for very long.

Polling a Completion Flag (Goetz1, p. 9-11)

```
// CalculatePrimes -- calculate as many primes as we can
// in ten seconds
public class CalculatePrimes extends Thread {
  public static final int MAX_PRIMES = 1000000;
  public static final int TEN_SECONDS = 10000;
  public volatile boolean finished = false;
  public void run() {
    int[] primes = new int[MAX_PRIMES];
    int count = 0;
    for( int i=2; count<MAX_PRIMES; i++ ) { // a polling loop
      // Check to see if the timer has expired
      if (finished) {
        break; // this thread stops looking for primes
      }
      // test i for primality ...
    }
  }
```

Polling example (cont.)

```
  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
    );
  }
} // end of CalculatePrimes
```

Responsiveness vs. efficiency in polling

- In `CalculatePrimes`, the finished flag is polled once for each integer `i` that is tested for primality. My evaluation:
  - This is a time-efficient design – the workers will spend most of their time testing for primality, with very little polling overhead.
  - This is a responsive design for smallish primes – a worker will execute at most a million instructions when testing a 5-digit prime number for primality, so it should "notice" the flag within a few milliseconds.
- If better responsiveness is required, the flag should be polled more frequently – making the polling less time-efficient…
  - Note that you must know a lot about the execution environment, in order to make a good tradeoff of accuracy for efficiency in polled code.
  - Ideally, the polling overhead is a few percent of total runtime. This optimises responsiveness without noticeably affecting runtime.
- "Keep it simple!" Polling is often an appropriate choice, even though it's not as elegant, efficient, or responsive as a more complex method.

Goetz's Prime-testing Task – my analysis

```
public void run() {
  int[] primes = new int[ MAX_PRIMES ];
  int count = 0;
  for ( int i=2; count<MAX_PRIMES; i++ ) {
    if ( finished ) { break; } // poll
    boolean prime = true;
    for ( int j=0; j<count; j++ ) { // test for primality
      if ( i % primes[j] == 0 ) {
        prime = false; break;
      }
    }
    // There are 78,498 primes less than MAX_PRIMES (= 1000000),
    // so the primality test should complete within a few msec.
    if ( prime ) {
      primes[ count++ ] = i;
      System.out.println( "Found prime: " + i );
} } }
```

Overhead of polling Goetz's flag

- It takes only a few CPU instructions to test a flag

```
if (finished) { break; }
```

  - Usual case: there is no extra delay on reading a volatile flag, when the thread already has read-privileges for that flag.
  - Occasionally: the thread doesn't yet have read-privileges, and must wait for a main-memory read (maybe a few microseconds).
  - Worst case: the worker thread must wait for the `main()` thread to finish its write.
    - This case is extremely rare, because Goetz's finished flag is written only once per program execution.
- My estimate: Goetz's workers spend
  - a few microseconds on each poll, and
  - a few milliseconds on each primality test when **MAX_PRIMES** = 1000000.
  - The code is probably bottlenecked on `println()`!

wait(), notify(), and notifyAll()

- Goetz1:"`wait()` causes the calling thread to sleep until
  - it is interrupted with `Thread.interrupt()`,
  - the specified timeout elapses, or
  - another thread wakes it up with `notify()` or `notifyAll()`.
- When `notify()` is invoked on an object,
  - if there are any threads waiting on that object via `wait()`, then one thread will be awakened.
- When `notifyAll()` is invoked on an object, all threads waiting on that object will be awakened.
- The `Object` class defines the methods `wait()`, `notify()`, and `notifyAll()`.
  - To execute any of these methods, you must be holding the lock for the associated object."

- For the CompSci 230 exam:
  - you should know that these methods exist, but their details are not examinable!

Usage Notes (Goetz)

- “These methods are the building blocks of more sophisticated locking, queuing, and concurrency code.
  - However, the use of `notify()` and `notifyAll()` is complicated.
  - In particular, using `notify()` instead of `notifyAll()` is risky.
  - Use `notifyAll()` unless you really know what you're doing.
- Rather than use `wait()` and `notify()` to write your own schedulers, thread pools, queues, and locks, you should
  - use the `util.concurrent` package (see Resources),
    - a widely used open source toolkit full of useful concurrency utilities.

Thread priorities

- Goetz: "The Thread API allows you to associate an execution priority with each thread.
  - However, how these are mapped to the underlying operating system scheduler is implementation-dependent.
  - In some implementations, multiple – or even all – priorities may be mapped to the same underlying operating system priority.
- Many people are tempted to tinker with thread priorities when they encounter a problem like deadlock, starvation, or other undesired scheduling characteristics.
  - More often than not, however, this just moves the problem somewhere else.
  - **Most programs should simply avoid changing thread priority."**

Goetz's warning about thread-safety

- **While the thread API is simple, writing thread-safe programs is not.**
- When variables are shared across threads,
  - you must take great care to
  - ensure that you have properly synchronized both read and write access to them.
- When writing a variable that may next be read by another thread, or reading a variable that may have been written by another thread,
  - you must use synchronization to ensure that changes to data are visible across threads.

Goetz' final warning

- When using synchronization to protect shared variables,
  - you must ensure that
  - not only are you using synchronization, but [also that]
  - the reader and writer are synchronizing on the same monitor.
- Furthermore,
  - if you rely on an object's state remaining the same across multiple operations, or
  - rely on multiple variables staying consistent with each other (or consistent with their own past values),
  - you must use synchronization to enforce this.
- But simply synchronizing every method in a class does not make it thread safe – **it just makes it more prone to deadlock**.

Goetz's summary

- Every Java program uses threads, whether you know it or not.
- If you are using either of the Java UI toolkits (AWT or Swing),
  - Java Servlets,
  - RMI, or
  - JavaServer Pages or
  - Enterprise JavaBeans technologies,
  - you may be using threads without realizing it.
- There are a number of situations where you might want to explicitly use threads to improve the performance, responsiveness, or organization of your programs. These include:
  - Making the user interface more responsive when performing long tasks
  - Exploiting multiprocessor systems to handle multiple tasks in parallel
  - Simplifying modeling of simulations or agent-based systems
  - Performing asynchronous or background processing

Learning Goals for Today

- Learn a little more Java:
  - `wait(), notify(), notifyAll().`
  - I do not expect you to be able to write code which invokes these methods appropriately.
  - The syntax is uncomplicated, but the code-design issues are very difficult.
- You *may* be examined on
  - Your understanding of the ways in which threads can safely signal each other, without "stepping on" each others' variables.
  - Your analysis of a multithreaded code, to determine whether or not there is some inappropriate interaction between its thread which may lead to deadlock or to corrupted computations.