Object - Oriented Programming & Design

PART XII: Java Threads

CSCI 4448

Java Threads

- A **Thread** is an execution process.
- Only one thread at a time may be "running" on a single processor machine at any given time.
- In an environment that supports multi-threading, significant efficiencies and design elegance can be gained by careful use of Threads (multiple concurrent flows of control through the program).
- Java provides excellent support for Threads, but there are significant holes.
- Multi-threaded programming is hard to get right.
- For a good Java reference, see *Concurrent Programming in Java - Design Principles and Patterns (2nd Ed.)*
  Doug Lea, Addison Wesley 1996. <ISBN 0-201-31009-0>

Java Threads (cont.)

Threads are indispensable …

- for user interfaces that must remain responsive while simultaneously computing some result (the Fractal Applet is an example of this).
- for servers with more than one simultaneous client.
- for polling loops (if necessary).
- for increased parallel-processing performance.
- when modeling a naturally concurrent, parallel, or asynchronous situation.

Note: Even if main() returns, a Java program will continue to run as long as one or more *non-daemon* threads remains alive. This is the case with most Java AWT applications.

Thread & Runnable

- Multi-Threaded programming is hard despite the **`java.lang.Thread`** class and the **`java.lang.Runnable`** interface, which have relatively easy syntax:

```
public interface Runnable
{
  public abstract void run();
}

public class Thread ...
{
  public Thread( Runnable runner ) { … }
  public synchronized void start() { … } // calls runner's run()
  ...
}
```

Simple Thread Example

```
class ThreadTest {
  public static void main( String[] args ) {
    Thread t = new Thread( new WorkerProcess( ) );
    System.out.print( "M1 : " );
    t.start( );
    System.out.print( "M2 : " );
  }
}
class WorkerProcess implements Runnable {
  public void run() {
    System.out.print( "Run ! " );
  }
}
```

- *Outputs:* `M1 : M2 : Run !`
- *Outputs:* `M1 : Run ! M2 :`

Thread Safety

- Interleaved operations (by multiple Threads) can *easily* corrupt data.
- Whenever 2 or more concurrent Threads call methods on behalf of the same object, care must be taken to ensure the integrity of that object.
- For example, a JVM may switch Threads in the middle of a 64-bit long or double assignment :-(. 64 bit operations are not considered to be *atomic*. If some other Thread attempts to use the intermediate value of the half-copied double, something bad will likely happen.

Thread **BUGS** can be hard to find and fix, especially in light of the following:

- The Java language does not guarantee Thread-switching *fairness*.
- Different JVMs use different Thread-switching algorithms, resulting in code that works on one platform but not on others.
- A single program may have different behavior on different runs.
- A program may crash one day after running correctly for long periods of time.

Thread Safety (cont.)

- Under what circumstances would this code not be "Thread-safe" ?!?

```
public class Foo {
  private int maxElements = 10;
  private int numElements = 0;
  private Vector elements = new Vector();

  public void add( Object newElement ) {
    numElements = elements.length();
    if( numElements < maxElements ) {
       elements.addElement( newElement );
} } }
```

- Two threads operating on one instance of this class at the same time; numElements = 9; both Threads get to point just before the **`if`** statement…
- How might we fix the problem?

Thread Safety (cont.)

- The key to achieving Thread ***safety*** lies in the concept of ***mutual exclusion***.
- Blocks of code in Java may be declared to be ***synchronized***. In order to enter a synchronized block of code, a Thread must acquire the key to a ***lock*** (aka: a "mutex") held by the block's so-called "***monitor object***." If the monitor object's lock is already held by another Thread, then the new Thread attempting to enter the block of code must wait (the JVM will automatically block it) until the lock is released. Any java.lang.Object may be such a synchronization lock monitor.
- Synchronization locks are only respected by synchronized blocks of code that use the same monitor object. In other words, just because a block of code is synchronized doesn't mean it is protected from concurrency problems. All other code which could possibly interfere with the state of the object in question must also be synchronized using the same monitor object. This can be tricky to get right.
- The synchronized statement acts as a gate to the subsequent block of code. To pass through the gate, the Thread must acquire the lock. Only one Thread at a a time is allowed to acquire any given lock. A single Thread may hold more than one lock at one time.
- Unlocked code is not protected.
- Another way to achieve mutual exclusion is to design the controller such that all processing that might operate on common resources is represented as some kind of event or command that can be put into a single ***queue***, with a single thread that runs the commands one at a time.

The Synchronized Keyword

```
public synchronized void foo() {
  bar();
}
```

- ***Is equivalent to:***

```
public void foo() {
  synchronized( this ) { // "this" is the monitor object
    bar();
} }
```

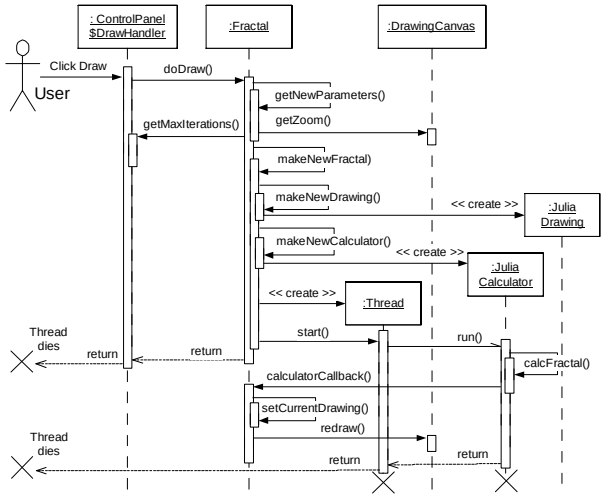
- If the above method were static, then the monitor object would be the single instance of class Class for the given class.

Class Thread

- Java provides various mechanisms for concurrent Threads to coordinate their efforts…
- Note: static methods operate on the *current* Thread.

```
class Thread implements Runnable
  public Thread( Runnable r )
  public static void sleep( long ms ) throws InterruptedException
  public static void yield()
  public static Thread currentThread()
  public synchronized void start()
  public final boolean isAlive()
  public final void join() throws InterruptedException
  public final void suspend() // deprecated.
  public final void resume() // deprecated.
  public final void stop()  // deprecated. have run() return.
  // . . .
```

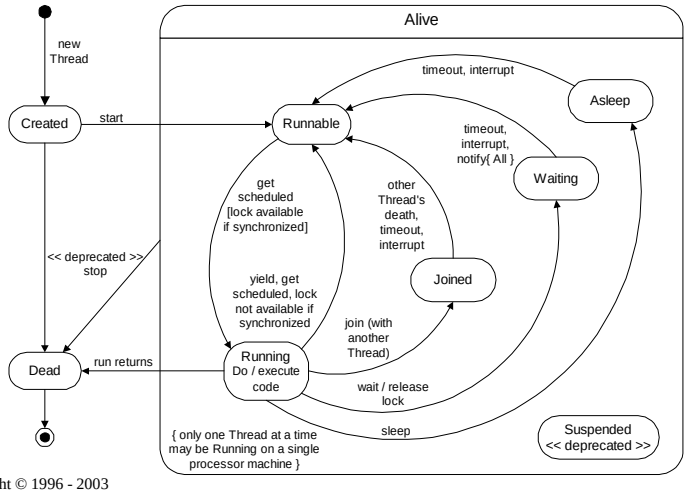
Example: Fractal Applet



Example: Fractal Applet (cont.)

- Based on information contained in the Fractal Applet Sequence Diagram, where is there a need for a synchronization strategy?
- Note the use of the open arrowhead in UML to indicate an *asynchronous* method call.
- Look at the source code for the Fractal Applet...
- Note the use of **`Thread.yield()`** in the inner loop of the calculator.
- Try commenting out the synchronized keyword for class Fractal's **`calculatorCallback()`** method and look for weird bugs (try creating a zoom rectangle just as a new drawing completes).
- Notice that calculator threads may be stopped.

Java Thread Life Cycle (State Diagram)



java.lang.ThreadDeath

```
class ThreadDeath extends Error { ... }
```

- This Exception is unique in that if it is caught, it must be *rethrown*, or else the Thread's resources do not get cleaned up; this includes releasing locks. It is for this reason that **`Thread.stop()`** is deprecated. ThreadDeath is to be avoided.

How to avoid Thread.stop

```
public class ThreadStop implements Runnable {
  private boolean stop = false;
  public static void main( String[] args ) {
    new ThreadStop().go();
  }
  private void go() {
    new Thread( this ).start();
    sleep( 100 );
    stop(); // Not the same as Thread.stop
  }
  public void run() {
    for( int i = 0 ; ! stop ; ) {
      System.out.println( "i = " + i++ );
      sleep( 10 );
    } // run() returns when stop == true
  }
```


How to avoid Thread.stop (cont.)

```
  private void sleep ( long milliseconds ) {
    try {
      Thread.sleep( milliseconds );
    }
    catch( InterruptedException ignore ) {}
  }

  public void stop() {
    stop = true;
  }
} // end ThreadStop
```

- *I have seen this code output a counter from 0-2 and also from 0-10 ...*

Join Example

- The join() method will block until a given thread dies. Meanwhile, if it has any synchronization locks, it does not release them...

```
public class JoinTest implements Runnable
{
  public static void main( String[] args ) {
    new JoinTest().go();
  }

  public synchronized void run() {
    System.out.println( "I am alive! " );
  }
```


Join Example (cont.)

```
  private void go() {
    Thread t = new Thread( this );
    System.out.println( "Thread t is alive: " + t.isAlive() );
    synchronized( this ) {
      t.start();
      System.out.println( "Thread t is alive: " + t.isAlive() );
    }
    try {
      t.join();
      System.out.println( "Thread t is alive: " + t.isAlive() );
    }
    catch( InterruptedException ie ) { }
  }
} // end JoinTest
```

Join Example (cont.)

Outputs:

```
Thread t is alive: false
Thread t is alive: true
I am alive!
Thread t is alive: false
```

- *What if the whole method* **`go()`** *were synchronized instead of just part of it?*
  The program would hang forever… why?
- *What if there were no synchronized statements at all?*
  The above behavior would not be guaranteed. How might it differ?

Wait & NotifyAll

- Any *java.lang.Object* can be used as a synchronization monitor, holding the lock for synchronized code. This functionality is built into the class Object.
- The class Object also provides a wait/notify service, allowing different Threads to coordinate their efforts (eg: producer / consumer).

```
class Object
{
  public final void wait() throws InterruptedException;
  public final void notify(); // usually use notifyAll()
  public final void notifyAll();
  // . . .
}
```

Wait & NotifyAll (cont.)

- The wait(), notify() and notifyAll() methods can only be called from within a synchronized block of code; the object on whose behalf they are called is the monitor object which holds the synchronization lock.
- The **wait()** method releases the lock, and then puts the current Thread into a wait state until some other Thread (that then holds the same lock) calls **notifyAll()**.
- Notify() will choose one arbitrary Thread that is waiting on the lock in question and force it out of the wait state into the runnable state. This might not be the Thread that you want! If you are unsure about this, use notifyAll(). Note however that there is a possible performance problem with notifyAll() - a "liveness" problem known as the "Thundering Herd".
- Wait() only releases one of the Thread's (possibly many) monitor locks. If there's more than one lock owned by the Thread, this can lead to a "lock out" condition if the Thread required to later call notifyAll() needs first to acquire one of the other unreleased locks.

Wait & NotifyAll (cont.)

- If a Thread is in the wait state, below, and some other Thread (running in code synchronized using bar as the monitor) sets conditionTrue and calls **bar.notifyAll**(), the waiting Thread will get bumped out of its wait state, allowing it to vie once again for bar's lock. Whenever it gets the lock, it will see conditionTrue, and will proceed to "do something" ...

```
class Foo {
  public void doSomethingAsSoonAsConditionTrue() {
    synchronized( bar ) {
      while( ! conditionTrue ) {
        try {
          bar.wait();
        }
        catch( InterruptedException ie ) { }
      }
      // "do something"
}  }  }
```

Deadlock & *Liveness*

- Once you ensure that your code is **Thread-safe** then there is still the problem of ensuring that the code always remains **alive**. Misuse of Java's built-in Threading facilities (examples: overuse of synchronization, naïve design) can cause serious performance degradations, **lock out**, & **deadlock** (you'll know it when you see it - your code just hangs… ;-(
- Deadlock is the condition where two Threads lock resources in different orders… Thread 1 locks resource A, then B; Thread 2 locks B, then A. Bad timing will cause this to hang forever.
- Deadlock is not possible if there is only one monitor object in use.
- Deadlock is not possible if locks are always acquired in the same A, B order.
- Java does not support timeouts on a synchronization block, and so Java deadlock detection is not feasible.
- Most commercial RDBMSs have deadlock detection, usually killing one Thread at random; it is the application programmer's responsibility to detect a SQLException and retry the failed transaction.
- ***Design*** your multi-threaded solution very carefully.

Deadlock & *Liveness (cont.)*

Other ***liveness*** problems include:

- synchronization is slow
- wait forever (notify never called) - program hangs
- thundering herd
- lock out - program hangs
- join with "immortal" Thread - program hangs
- unfair time slicing
- mystery bugs that are really safety problems at their core

Using a Queue

- Consider the design option of using a queue to effectively serialize all potentially contentious work (commands & events) using a single "queue thread." This can be an effective design, sidestepping many thread related difficulties.
- For multi-threaded GUIs using Swing, note that the AWT event / paint dispatching thread can be "hijacked" for this purpose with a single line of code:

```
SwingUtilities.invokeLater( runnable ); // javadoc on-line
```

- Consider a GUI application that uses 3 threads: 1) the main AWT event / paint thread, 2) an animation loop, 3) a polling loop that looks for new data to appear. All 3 of these threads will want to draw something to the screen, and so there is potential conflict. To eliminate the possibility of thread bugs here, the animation loop thread can put "next frame" commands onto the AWT queue, and the polling loop can put "new data" commands onto the queue, using **`SwingUtilities.invokeLater().`**
- In other words, whenever worker threads need to call back to the main controller, they do so by putting a command onto the queue, thus achieving mutual exclusion.
- The same idea can be applied to servers, but one would have to use a different thread-safe queue implementation. Variations on this idea allow for increased concurrency.

Thread Summary

- The first priority is to prevent safety violations, such as corruption caused by interleaved operations, by designing a synchronization locking strategy.
- The simplest strategy is to design your code such that multiple Threads do not operate on the same object(s) at the same time (no synchronization required).
- Choose your synchronization monitor objects carefully. If you can get away with using a single monitor object, you will prevent deadlock.
- In Model-View-Controller GUIs, it is possible to design things in such a way so that all synchronization is done at the controller.
- Misuse (or overuse) of synchronized code can lead to liveness problems.
- Don't assume anything about Thread time-slicing.
- Avoid "polling" Threads if you can use the Observer design pattern.
- Consider the design option of using a queue to effectively serialize work (commands & events) using a single "queue thread."

Thread Summary (cont.)

- Testing and debugging can be difficult due to the lack of repeatability and platform variation. There are some helpful "threadalizer" tools out there...
- There are numerous "patterns" for dealing with Thread problems (such as using a single-threaded queue to serialize work); study them.
- We have only scratched the surface…
- Further coverage of Java Threads is beyond the scope of this course.
- Avoid over-using threads; consider design alternatives.

Performance optimizations:

- Synchronize the smallest possible block of code to minimize the odds of multi-Thread contention.
- Don't synchronize the methods that are called only from one thread.
- If you must have many threads, consider recycling them with a ThreadPool.

Vocabulary

• Thread / Runnable
• interleaved operations
• bug types: safety / liveness
• atomicity
• asynchronous vs. synchronous
• JVM thread-switching / fairness
• synchronized keyword
• deadlock & "lock out"
• synchronization strategy
• Java Thread methods: start / sleep / yield / join / isAlive
• monitor methods: wait / notify / notifyAll
• Concurrency control
• ThreadDeath (extends Error) // don't call Thread.stop()
• Producer Consumer pattern
• Single-Threaded Queue pattern / SwingUtilities.invokeLater(r);
• Not covered: Thread Groups, the volatile keyword, Daemon Threads