

Object - Oriented Programming & Design

PART XII: Java Threads

Copyright © 1996 - 2008

Ron LeMaster & David Leberknight. All rights reserved.

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 that doesn’t make it easy!
- Multi-threaded programming is DIFFICULT.
- For a good low-level Java reference, see *Concurrent Programming in Java - Design Principles and Patterns (2nd Ed.)*
Doug Lea, Addison Wesley 1996. <ISBN 0-201-31009-0>
- Now, with **java.util.concurrent** !!! *Java Concurrency in Practice*
Brian Goetz, Addison Wesley 2006. <ISBN 0-321-34960-1 >

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 2

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 web servers.
- 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.

Note: Even if your program never explicitly creates a thread, frameworks do. This is not a topic you can safely ignore.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 3

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()
    ...
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 4

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 :

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 5

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, it's a bug.

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.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 6

Thread Safety (cont.)

Code that must be *thread safe*:

- Anything shared between two or more threads
- Singletons
- Globals that have state and not just behavior

A class is thread safe if it never enters an invalid state, even when an instance of that class is accessed by multiple threads concurrently.

Objects that are always thread safe:

- Objects that have no state
- Objects that are immutable (final)
- Objects that have no *compound* operations (ie: every change in state occurs *atomically*)

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 7

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 ArrayList elements = new ArrayList();

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

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

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 8

Mutual Exclusion

- 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 lock’s “*monitor object*.” If the lock is already held by another Thread, then the new Thread must wait (the JVM will block it) until the lock is released. Any `java.lang.Object` may be used as a 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’s 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. In other words, use only one lock.
- 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 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.
- Java’s low level mechanisms to control threads must be used in the context of higher level policies, with discipline and understanding.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 9

Mutual Exclusion (cont.)

Another way to achieve *mutual exclusion* is to design the application such that all processing that might operate on shared resources is represented as some kind of event or command that can be put into a *queue*, with a single thread that runs the commands one at a time. Solve the problem by avoiding it altogether! This is an example of the producer-consumer design pattern.

- For applications in Java that use Swing, all code that updates UI widgets should do so using the single Event / Paint Dispatching Thread (Swing widgets are NOT thread safe). This can be achieved with a single line of code:

```
SwingUtilities.invokeLater( runnable );
```

Performance optimizations:

- Synchronize the smallest possible block of code to minimize the odds of multi-Thread contention.
- Don’t synchronize methods that are called only from one thread.
- Don’t use synchronized java library classes unless you need to (they’re slow). For example, use `StringBuilder`, not `StringBuffer`.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 10

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 instance of class `Class` for the given class.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 11

Class Thread

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

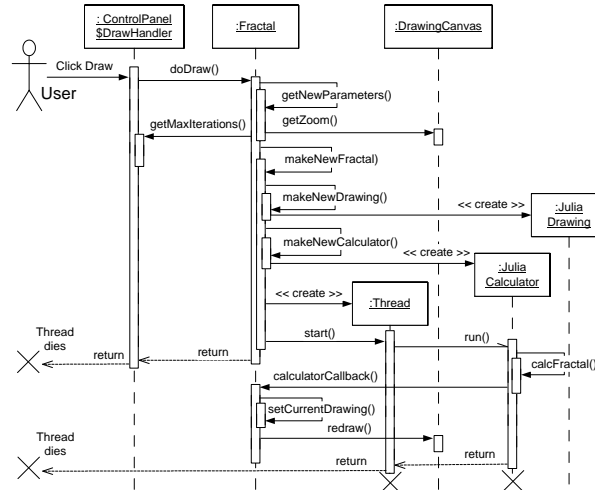
```
class Thread implements Runnable  
    public Thread( Runnable r )  
    public static void sleep( long ms ) throws InterruptedException  
    public static void yield()  
    public static Thread currentThread()  
    public synchronized void start()  
    public final boolean isAlive()  
    public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

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 12

Example: Fractal Applet



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 13

Example: Fractal Applet (cont.)

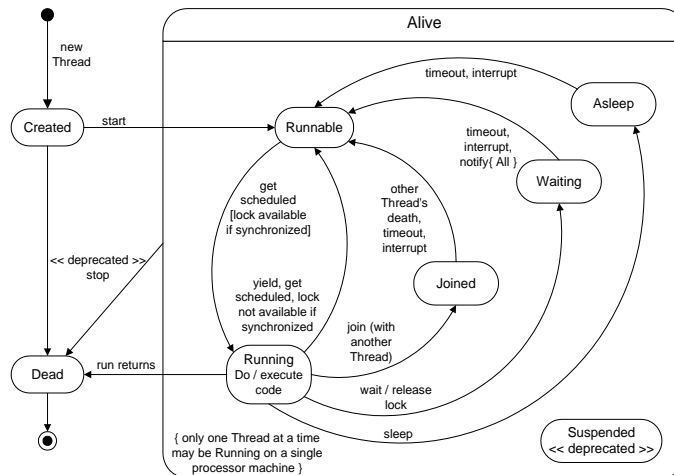
- Based on information contained in the Fractal Applet Sequence Diagram, where is there a need for a synchronization strategy?
- Note the use of the open arrowhead in UML to indicate an *asynchronous* method call.
- Look at the source code for the Fractal Applet...
- Note the use of `Thread.yield()` in the inner loop of the calculator. This is not guaranteed to do anything. But for most browsers it does help the UI responsiveness when users click Next and Previous to browse existing fractal images concurrent with a new image being created.
- 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.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 14

Java Thread Life Cycle (State Diagram)



Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 15

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.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 16

How to avoid Thread.stop

```
public class ThreadStop implements Runnable {
    private volatile boolean stop = false; // Note: volatile
    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
    }
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 17

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 ...

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 18

Volatile and the Java Memory Model

- Without the `volatile` keyword, this program could in theory run forever! The reasoning has to do with the “Java memory model” whereby threads that access shared variables may keep private working copies of the variables; the `volatile` keyword forces all updates to the variable to be pushed out to shared main memory. Otherwise this is only guaranteed to occur at synchronization points.
- Synchronization ensures more than mutual exclusion, it ensures up-to-date visibility of shared memory.

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
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 21

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?

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 22

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 *Thread*s 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();
    // . . .
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 23

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.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 24

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"
        } } }
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 25

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.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 26

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

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 27

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.”
- 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, so there is potential conflict. To eliminate the possibility of thread bugs, 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()**.
- This *producer-consumer* design can be applied to servers.
Use `java.util.concurrent.BlockingQueue`.
- Variations on this idea allow for increased concurrency.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 28

Collection Class Synchronization Wrappers

- This is an example of the decorator design pattern.
- The monitor object for synchronization is `sychMap`.

```
Map normalMap = new HashMap();
Map sychMap = Collections.synchronizeMap( normalMap );
sychMap.put( "foo", "bar" ); // thread safe
Iterator it = sychMap.keySet().iterator();
while( it.hasNext )
{
    String key = (String) it.next();
    // Might throw ConcurrentModificationException
    // ...
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 29

Collection Class Synchronization Wrappers

- One way to prevent `ConcurrentModificationException` is to hold the lock for the duration of the iteration. But it is not generally a good idea to hold locks for long durations.

```
Map normalMap = new HashMap();
Map sychMap = Collections.synchronizeMap( normalMap );
sychMap.put( "foo", "bar" ); // thread safe
synchronized( sychMap )
{
    Iterator it = sychMap.keySet().iterator();
    while( it.hasNext )
    {
        // complete batch operations on the map's contents
    }
}
```

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 30

Java.util.concurrent

Consider replacing collection class synchronization wrappers with *concurrent collections*. They are more sophisticated, and can offer dramatic scalability improvements.

- Study `java.util.concurrent.*`
- `java.util.concurrent.atomic.AtomicLong`
- `BlockingQueue`
- `ConcurrentHashMap`
- `CopyOnWriteArrayList`
- `Semaphores`, `Barriers`, `Latches` (synchronizers)
- `FutureTask`
- `ExecutorService`
- `ThreadFactory`

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 31

Thread Summary

- The first rule of thumb is to try to AVOID thread problems by not sharing state, and by using (final) immutable variables.
- Consider the design option of using a queue to serialize “commands” using a single “consumer thread.”
- Try to design your code so that multiple threads do not operate on the same object(s) at the same time.
- When designing a synchronization locking strategy to prevent safety violations, choose your monitor objects carefully. If you can get away with using a single monitor object, you will prevent deadlock.
- 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.
- If you must have many threads, consider recycling them with a `ThreadPool`.

Copyright © 1996 - 2008

David Leberknight & Ron LeMaster. All rights reserved.

(XII) Java Threads - 32

Thread Summary (cont.)

- Document thread safety. If a class is required to be thread safe, say so!
- 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.
- Read *Java Concurrency in Practice*
Brian Goetz, Addison Wesley 2006. <ISBN 0-321-34960-1 >