Taming Java™
Programming Language
Threads (“Java Threads”)

Allen I. Holub
President
Holub Associates
www.holub.com  allen@holub.com
What We’ll Do Today

- Programming threads in the Java™ programming language is fraught with peril, but is mandatory in a realistic program
- This talk discusses traps and pitfalls, along with some solutions
- This talk focuses on material not covered in most books
Shameless Self Promotion

• Former CTO, NetReliance
• Learned threads doing real-time programming
• Talk based on my JavaWorld™ “Java Toolbox” column, now a book:
  – Taming Java™ Threads
    (Berkeley: APress, 2000; http://www.apress.com)
• Source code, etc., found at http://www.holub.com
• My Prejudices and Bias
  – I do not work for Sun
  – I have opinions and plan to express them.
    The appearance of impartiality is always just appearance
  – Java™ technology is the best thing since sliced bread
    (but bakery bread is better than sliced)
I’m Assuming That…

• I’m assuming you know:
  – The language, including inner classes
  – How to create threads using Thread and Runnable
  – synchronized, wait(), notify()
  – The methods of the Thread class

• You may still get something out of the talk if you don’t have the background, but you’ll have to stretch
We’ll Look At

- Thread creation/destruction problems
- Platform-dependence issues
- Synchronization & Semaphores (synchronized, wait, notify, etc.)
- Memory Barriers and SMP problems
- Lots of other traps and pitfalls
- A catalog of class-based solutions
- An OO-based architectural solution
Books, Etc.

Allen Holub  *Taming Java™ Threads*
- Berkeley, APress, 2000

Doug Lea  *Concurrent Programming in Java™: Design Principles and Patterns*, 2nd Edition
- Reading: Addison Wesley, 2000

Scott Oaks and Henry Wong  *Java™ Threads*
- Sebastopol, Calif.: O’Reilly, 1997

Bill Lewis and Daniel J. Berg  *Threads Primer: A Guide to Multithreaded Programming*

http://developer.java.sun.com/developer/technicalArticles/Threads/
Words to Live By

All nontrivial applications for the Java™ platform are multithreaded, whether you like it or not.

It’s not okay to have an unresponsive UI. It’s not okay for a server to reject requests.
Threads vs. Processes

- A **Process** is an address space
- A **Thread** is a flow of control through that address space
  - Threads share the process’s memory
  - Thread context swaps are much lower overhead than process context swaps
Threads vs. Processes in the Java Programming Language

- A process is a JVM™ instance
  - The Process contains the heap (everything that comes from `new`)
  - The heap holds all static memory
- A thread is a runtime (JVM™) state
  - The “Java Stack” (runtime stack)
  - Stored registers
  - Local variables
  - Instruction pointer
- **Thread-safe** code can run in a multithreaded environment
  - Must synchronize access to resources (e.g., memory) shared with other threads or be reentrant
  - Most code in books isn’t thread safe
Thread Behavior Is Platform Dependent!

- You need to use the OS threading system to get parallelism (vs. concurrency)

- Different operating systems use different threading models (more in a moment)
- Behavior often based on timing
- Multithreaded apps can be slower than single-threaded apps (but be better organized)
Priorities

- The Java programming language has 10 levels
- The Solaris™ OS has 231 levels
- NT offers 5 (sliding) levels within 5 “priority classes”

\[ \begin{array}{cccccccccccccccccccc}
\hline
\text{Idle} & \text{Normal} & \text{Normal} & \text{High} & \text{Real-Time} \\
\end{array} \]

- NT priorities change by magic
  - After certain (unspecified) I/O operations priority is boosted
    (by an indeterminate amount) for some (unspecified) time
  - Stick to \text{Thread.MAX\_PRIORITY}, \text{Thread.NORM\_PRIORITY}, \text{Thread.MIN\_PRIORI}
Threading Models

- **Cooperative** (Windows 3.1)
  - A Thread must voluntarily relinquish control of the CPU
  - Fast context swap, but hard to program and can’t leverage multiple processors

- **Preemptive** (NT)
  - Control is taken away from the thread at effectively random times
  - Slower context swap, but easier to program and multiple threads can run on multiple processors

- **Hybrid** (Solaris OS, Posix, HPUX, Etc.)
  - Simultaneous cooperative and preemptive models are supported
NT Threading Model

(Win32 “fibers” are so poorly documented, and so buggy, they are not a real option)
Solaris™ OS Threading Model
Do Not Assume a Particular Environment

• Assume both of these rules, all the time:
  1. A thread can prevent other threads from running if it doesn’t occasionally yield
     • By calling yield(), performing a blocking I/O operation, etc.
  2. Thread can be preempted at any time by another thread
     • Even by one that appears to be lower priority than the current one
Thread Creation

• Java technology’s Thread class isn’t (a thread)
  – It’s a thread controller

class Operation implements Runnable
{
    public void run()
    {
        // This method (and the methods it calls) are
        // the only ones that run on the thread.
    
    }
}

Thread thread_controller = new Thread(new Operation);
thread_controller.start();
Java Threads Aren’t Object Oriented (1)

- Simply putting a method in a `Thread` derivative does not cause that method to run on the thread
  - A method runs on a thread only if it is called from `run()` (directly or indirectly)

```java
class Fred extends Thread {
    public void run() {
        // This method (and the methods it calls) are
        // the only ones that run on the thread.
    }

    public foo() {
        // This method will not run on the thread since
        // it isn’t called by `run()`
    }
}
```
Java Threads Aren’t Object Oriented (2)

- Objects do not run on threads, methods do
- Several threads can send messages to the same object simultaneously
  - They execute the same code with the same this reference, so share the object’s state
Basic Concepts: Atomic Operations (Atomicity)

- Atomic operations can’t be interrupted (divided)
- Assignment to double or long is not atomic

```java
long x;
thread 1:
    x = 0x0123456789abcdef
thread 2:
    x = 0;
possible results:
    0x0123456789abcdef;
    0x0123456700000000;
    0x0000000089abcdef;
    0x0000000000000000;
```

64-bit assignment is effectively implemented as:

```java
x.high = 0x01234567
x.low = 0x89abcdef;
You can be preempted between the assignment operations.
```
Basic Concepts: Synchronization

• Mechanisms to assure that multiple threads:
  – Start execution at the same time and run concurrently ("condition variables" or "events")
  – Do not run simultaneously when accessing the same object ("monitors")
  – Do not run simultaneously when accessing the same code ("critical sections")

• The **synchronized** keyword is essential in implementing synchronization, but is poorly designed
  – E.g., No timeout, so deadlock detection is impossible
Basic Concepts: Semaphores

• A semaphore is any object that two threads can use to synchronize with one another
  – Don’t be confused by Microsoft documentation that (incorrectly) applies the word “semaphore” only to a Dijkstra counting semaphore

• Resist the temptation to use a Java native interface (JNI) call to access the underlying OS synchronization mechanisms
The Mutex (Mutual-exclusion Semaphore)

- The mutex is the key to a lock
  - Though it is sometimes called a “lock”
- Ownership is the critical concept
  - To cross a `synchronized` statement, a thread must have the key, otherwise it blocks (is suspended)
  - Only one thread can have the key (own the mutex) at a time
- Every Object contains an internal mutex
  ```java
  Object mutex = new Object();
  synchronized( mutex )
  {
  // guarded code is here.
  }
  ```
  - Array are also objects, as is the `Class` object
Monitors and Airplane Bathrooms

• A **monitor** is a body of code (not necessarily contiguous), access to which is guarded by a single mutex
  – Every object has its own monitor (and its own mutex)

• Think “airplane bathroom”
  – Only one person (thread) can be in it at a time (we hope)
  – Locking the door acquires the associated mutex—You can’t leave without unlocking the door
  – Other people must line up outside the door if somebody’s in there
  – Acquisition is not necessarily FIFO order
Synchronization With Individual Locks

- Enter the monitor by passing over the synchronized keyword
- Entering the monitor does not restrict access to objects used inside the monitor—it just prevents other threads from entering the monitor

```java
long field;
Object lock = new Object();

synchronized(lock)
{
    field = new_value
}
```
Method-level Synchronization

class Queue
{
    public synchronized void enqueue(Object o)
    {
        /*...*/
    }
    public synchronized Object dequeue()
    {
        /*...*/
    }
}

- The monitor is associated with the object, not the code
  - Two threads can happily access the same synchronized code at the same time, provided that different objects receive the request
  - E.g., Two threads can enqueue to different queues at the same time, but they cannot simultaneously access the same queue
  - Same as synchronized(this)
He Came in the Bathroom Window

- The Bathroom can have several doors

```java
class Bathroom_window {
    private double guard_this;

    public synchronized void ringo(double some_value) {
        guard_this = some_value;
    }

    public double george() { // WRONG! Needs synchronization
        return guard_this;
    }
}
```
Constructors Can’t Be Synchronized, So Always Have Back Doors

```java
class Unpredictable {
    private final int x;
    private final int y;

    public Unpredictable(int init_x, int init_y) {
        new Thread() {
            public void run() {
                System.out.println("x=" + x + " y=" + y);
            }
        }.start();

        x = init_x;
        y = init_y;
    }
}

• Putting the thread-creation code at the bottom doesn’t help (the optimizer might move it)
```
class Predictable
{
    private final int x;
    private final int y;

    public Predictable(int init_x, int init_y)
    {
        synchronized( this )
        {
            new Thread()
            {
                public void run()
                {
                    synchronized( Predictable.this )
                    {
                        System.out.println("x=\"+x+\" y=\"+y);
                    }
                }
            }.start();
            x = init_x;
            y = init_y;
        }
    }
}
class Synch
{
    synchronized int locking ( int a, int b )
    {
        return a + b;
    }
    int not_locking ( int a, int b )
    {
        return a + b;
    }
    static public void main(String[] arguments)
    {
        double start = new Date().getTime();
        for(long i = 1000000; --i >= 0 ;)
            tester.locking(0,0);
        double end = new Date().getTime();
        double locking_time = end - start;
        // repeat for not_locking
    }
}
Synchronization Isn’t Cheap

% java -verbose:gc Synch
Pass 0: Time lost: 234 ms.  121.39% increase
Pass 1: Time lost: 139 ms.  149.29% increase
Pass 2: Time lost: 156 ms.  155.52% increase
Pass 3: Time lost: 157 ms.  155.87% increase
Pass 4: Time lost: 157 ms.  155.87% increase
Pass 5: Time lost: 155 ms.  154.96% increase
Pass 6: Time lost: 156 ms.  155.52% increase
Pass 7: Time lost: 3,891 ms.  1,484.70% increase
Pass 8: Time lost: 4,407 ms.  1,668.33% increase

—200MHz Pentium, NT4/SP3, JDK 1.2.1, HotSpot 1.0fcs, E

• Contention in last two passes (Java Hotspot™ VM can’t use atomic-bit-test-and-set)
Synchronization Isn’t Cheap

But...

- The cost of stupidity is always higher than the cost of synchronization
  - Pick a fast algorithm
- Overhead can be insignificant when the synchronized method is doing a time-consuming operation
  - But in OO systems, small synchronized methods often chain to small synchronized methods
Avoiding Synchronization

• Reentrant code doesn’t need to be synchronized
  – Code that uses only local variables and arguments (no static variables, no fields in the class)

• Atomic operations do not need to be synchronized, but beware of reordering
  – Assignment to all non-64-bit things, including booleans and references are usually safe, but sequence not preserved
  – Must be declared volatile, but volatile might not work
  – Assignment to volatile doubles and floats should be atomic (but most VMs don’t do it)
  – Code may be reordered, so assignment to several atomic variables must be synchronized

• Sequence of volatile operations should be preserved, but usually isn’t
Avoiding Synchronization

- Synchronize the smallest block possible to minimize the odds of contention
  - Method-level synchronization should be avoided in very-high-performance systems

- Don’t synchronize the methods of classes that are called only from one thread
  - Use Collection-style synchronization decorators when you need synchronized behavior

```java
Collection c = new ArrayList();
c = Collections.synchronizedCollection(c);
```
Avoiding Synchronization

• Don’t access synchronized methods from synchronized methods
  – Synchronize public methods—Don’t synchronize private ones
  – Don’t use protected
  – Avoid Vector and Hashtable in favor of Collection and Map derivatives
  – Don’t use BufferedInputStream, BufferedOutputStream, BufferedReader, or BufferedWriter unless the stream is shared between multiple threads
    • You can use InputStream’s read(byte[])
    • You can roll your own decorators
Immutable Objects

- Synchronization not required (all access read-only)
- All fields of the object are final (e.g., String)
  - *Blank* finals are `final` fields without initializers
  - Blank finals must be initialized in all constructors

```java
class I_am_immutable
{
    private final int some_field;
    public I_am_immutable(int initial_value)
    {
        some_field = initial_value;
    }
}
```
- Might not compile with inner classes (there’s a long-standing compiler bug)

- Immutable ≠ constant (but it must be constant to be thread safe)
  - A `final` reference is constant, but the referenced object can change state
  - Language has no notion of “constant”, so you must guarantee it by hand
**Critical Sections**

- A critical section is a body of code that only one thread can enter at a time.
- Do not confuse a critical section with a monitor:
  - The monitor is associated with an object.
  - A critical section guards code.
- The easiest way to create a critical section is by synchronizing on a static field.

```java
static final Object critical_section = new Object();
synchronized( critical_section ) {
    // only one thread at a time
    // can execute this code
}
```
class Flintstone
{
    public void fred()
    {
        synchronized( Flintstone.class )
        {
            // only one thread at a time
            // can execute this code
        }
    }

    public static synchronized void wilma()
    {
        // synchronizes on the same object
        // as fred().
    }
}
Class vs. Instance Variables

- All **synchronized** static methods synchronize on the **same** monitor

- Think **class variables vs. instance variables**:
  - The class (static) variables and methods are effectively members of the Class object
  - The class (static) variables store the state of the class as a whole
  - The class (static) methods handle messages sent to the class as a whole
  - The instance (non-static) variables store the state of the individual objects
  - The instance (non-static) methods handle messages sent to the individual objects
class Foo
{
    static long x = 0;
    synchronized static void set_x( long x )
    {
        this.x = x;
    }
    synchronized /* not static */ double get_x()
    {
        return x;
    }
}

Thread 1:  Thread 2:
Foo o1 = new Foo();  Foo.set_x(-1);
long x = o1.get_x();

Results are undefined. (There are two locks here, one on the class object and one on the instance.)
1. Access all static fields through synchronized static methods, even if the accessor is a method of the class that contains the field

```java
class Okay {
    private static long unsafe;
    private static synchronized get() {
        return unsafe;
    }
    private static synchronized set(long x) {
        unsafe = x;
    }

    public /*not static*/ void foo(long x) {
        //...
        set(x);
    }
}
```
2. Synchronize explicitly on the class object when accessing a static field from an instance method

class Okay
{
    private static long unsafe;
    public void foo(long x)
    {
        //...
        synchronized( Okay.class )
        {
            unsafe = x;
        }
    }
}
3. Encapsulate all static fields in an inner class and provide exclusive access through synchronized methods of the inner class

class Okay
{
    private class Class_Variables
    {
        private long unsafe;
        public synchronized void do_something(long x)
        {
            unsafe = x;  // . . .
        }
    }
    static Class_Variables statics =
        new Class_Variables();
    public foo(long x)
    {
        statics.do_something( x );
    }
}
Singletons (One-of-a-kind Objects)

• Singletons often use critical sections for initialization

    public final class Singleton
    {
        static{new JDK_11_unloading_bug_fix(Std.class);} // prevent creation by new

        private static Singleton instance;
        private Singleton(){}  // prevent creation by new

        public synchronized static Singleton instance()
        {
            if( instance == null )
                instance = new Singleton();
            return instance;
        }
    }

    Singleton s = Singleton.instance()
Avoiding Synchronization in a Singleton by Using Static

• A degraded case, avoids synchronization

```java
public final class Singleton
{
    static{ new JDK_11_unloading_bug_fix(Std.class); }
    private Singleton(){}

    private static final Singleton instance = new Singleton();

    public /*unsynchronized*/ static Singleton instance()
    { return instance; }
}
```
Or Alternatively…

- Thread safe because VM loads only one class at a time and method can’t be called until class is fully loaded and initialized

- No way to control constructor arguments at run time

```java
public final class Singleton
{
    private static Singleton instance;
    private Singleton()
    {
    
        static{ instance = new Singleton(); } 

        public static Singleton instance()
        {
            return instance;
        }
    }
}
```
While We’re on the Subject…

In the JDK™ 1.1 release, all objects not accessible via a local-variable or argument were subject to garbage collection.

```java
public class JDK_11_unloading_bug_fix
{
    public JDK_11_unloading_bug_fix(final Class keep)
    {
        if (System.getProperty("java.version")
            .startsWith("1.1") )
        {
            Thread t = new Thread()
            {
                public void run()
                {
                    Class singleton_class = keep;
                    synchronized(this)
                    {
                        try{ wait();}
                        catch(InterruptedException e){}
                    }
                }
            }
            t.setDaemon(true);
            t.start();
        }
    }
}
```
Condition Variables

- All objects have a “condition variable” in addition to a mutex
  - A thread blocks on a condition variable until the condition becomes true
  - In the Java™ environment, conditions are “pulsed”—condition reverts to false immediately after waiting threads are released

- `wait()` and `notify()` use this condition variable
wait and notify Have Problems

• Implicit condition variables don’t stay set!
  – A thread that comes along after the `notify()` has been issued blocks until the next `notify()`

• `wait()` does not tell you if it returned because of a timeout or because the wait was satisfied (hard to solve)

• There’s no way to test state before waiting

• `wait()` releases only one monitor, not all monitors that were acquired along the way (nested monitor lockout)
class Notifying_queue
{
    private static final queue_size = 10;
    private Object[] queue = new Object[queue_size];
    private int head = 0;
    private int tail = 0;

    public void synchronized enqueue( Object item )
    {
        queue[++head %= queue_size] = item;
        this.notify();
    }

    public Object synchronized dequeue( )
    {
        try
        {
            while( head == tail ) // <-- MUST BE A WHILE
            {
                this.wait(); // (NOT AN IF)
            }
        }
        catch( InterruptedException e )
        {
            return null; // wait abandoned
        }

        return queue[++tail %= queue_size ];
    }
}
synchronized enqueue(. . .) ___ this.mutex.acquire();
{
    this.notify();  __________________ this.condition.set_true();
} _________________________ this.mutex.release();

synchronized dequeue(. . .) ___ this.mutex.acquire();
{
    while( head == tail ) __________ while( head==tail )
        this.wait(); ______________________ this.mutex.release();
    this.condition.wait_for_true();
    this.mutex.acquire();
}
________________________________________ this.mutex.release()
Condition Variables—Wait Is Not Atomic (2)

```java
synchronized enqueue(...)
{
    this.notify();                        this.condition.set_true();
}                                 this.mutex.release();

synchronized dequeue(...)
{
    while( head == tail )
        this.wait();                    this.condition.wait_for_true();
                                        this.mutex.acquire();
}                                     this.mutex.release();
```
synchronized enqueue(...) {
    this.notify();
    this.condition.set_true();
} this.mutex.release();

synchronized dequeue(...) {
    while (head == tail) {
        this.wait();
        this.condition.wait_for_true();
        this.mutex.acquire();
    }
} this.mutex.release()
Condition Variables—Wait Is Not Atomic (4)

synchronized enqueue(...) ___ this.mutex.acquire();
{
  this.notify();  
  this.condition.set_true();  
}  
  this.mutex.release();

synchronized dequeue(...) ___ this.mutex.acquire();
{
  while( head == tail )  
    this.wait();  
    this.mutex.release();  
    this.condition.wait_for_true();  
    this.mutex.acquire();  
}  
  this.mutex.release()
synchronized enqueue(...) {
    this.notify();
}  
this.condition.set_true();

synchronized dequeue(...) {
    while (head == tail) {
        this.wait();
    }
this.condition.wait_for_true();
this.mutex.acquire();
}

this.mutex.release();
Summarizing `wait()` Behavior

- `wait()` doesn’t return until the notifying thread gives up the lock
- A condition tested before entering a `wait()` may not be true after the `wait` is satisfied
- There is no way to distinguish a timeout from a `notify()`
Beware of Symmetric Multi-Processing (SMP) Environments

- The CPU does not access memory directly
- CPU read/write requests are given to a “memory unit,” which actually controls the movement (at the hardware level) of data between the CPU and main memory store
Some Common Memory Operations Are Inefficient

- Processors supporting a “relaxed memory model” can transfer blocks of memory between cache and the main memory store in undefined order!

- Consider:

```
int a[] = new int[10];
int b[] = new int[10];
for( int i = 0; i < a.length; ++i )
    b[i] = a[i];
```
Presto Chango!

- The memory unit notices the inefficiency and rearranges the requests!

\[
\text{CPU1} \rightarrow R_{a[0]} \rightarrow R_{a[1]} \rightarrow R_{a[n]} \rightarrow W_{b[0]} \rightarrow W_{b[1]} \rightarrow W_{b[n]} \rightarrow \text{MU1} \rightarrow \text{memory}
\]

- To produce:

\[
\text{CPU1} \rightarrow R_{a[0..n]} \rightarrow W_{b[0..n]} \rightarrow \text{MU1} \rightarrow \text{memory}
\]

- This change is good—it speeds memory access
BUT…

• The order in which changes are made in the source code may not be preserved at run time!
Don’t Panic

• Reordering doesn’t matter in single-threaded systems

• Reordering not permitted across “memory barriers” (effectively inserted around synchronized access)
Memory Barriers Are Created Indirectly by Synchronization

- **synchronized** is implemented using a memory barrier
  - So modifications made within a **synchronized** block will not move outside that block

- **volatile** should force memory synchronization, but don’t count on it
  - But might add access inefficiencies
  - JVM implementation of volatile is spotty—some don’t implement it at all
Avoiding Synchronization (Revisited)

- You **cannot** use `volatile` fields (e.g., boolean) to guard other code

```java
class I_wont_work {
    private volatile boolean okay = false;
    private long field = -1;
    //...
    public /*not synchronized*/ void wont_work() {
        if (okay) {
            // Might be -1.
            do something(field);
        }
    }
    public /*not synchronized*/ void enable() {
        field = 0;
        okay = true;
    }
}
```
Even Worse

• Memory modifications made in the constructor may not be visible, even though the object is accessible!

class Surprise{
    public long field;
    // . . .
    public Surprise()
    {
        field = -1;
    }
}

Thread 1:
    Surprise s = new Surprise();

Thread 2:
    System.out.println(s.field);

Modification of s might become visible before modification of field if memory unit rearranges operations.
Synchronization Can Fix Things

• This works

Object lock = new Object();

Thread 1:
synchronized( lock )
{ Surprised s = new Surprised();
}

Thread 2:
synchronized( lock )
{ System.out.println(s.get_field());
}
But Then Again, Maybe Not

• This might not work

```java
class Might_work {
    public long field;
    //...
    public Might_work() {
        synchronized (this) {
            field = -1;
        }
    }
    public synchronized get_field() {
        return field;
    }
}
Thread 1:
    Might_work m = new Might_work();
Thread 2:
    System.out.println(m.get_field());
```
Implicit assignment of zero to `field` is not inside the synchronized block. Modification of 0 to –1 may not be visible in `get_field()`.
### Double-checked Locking Doesn’t Work!

- **Is unreliable** even in single-CPU machine

```java
public final class Singleton
{
    static { new JDK_11_unloading_bug_fix(Std.class); }

    private static Singleton instance;
    private Singleton() {} // prevent creation by new

    public static Singleton instance()
    {
        if (instance == null)
        {
            synchronized (Singleton.class)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```
“Rules to Live By” in an SMP Environment (Gotchas)

• To assure that shared memory is visible to two threads: the writing thread must give up a lock that is subsequently acquired by the reading thread

• Modifications made while sleeping may not be visible after sleep() returns

• Operations are not necessarily executed in source-code order (not relevant if code is synchronized)

• Modifications to memory made after a thread is created, but before it is started, may not be visible to the new thread
“Rules to Live By” in an SMP Environment (Things That Work)

- Modifications made by a thread before it issues a `notify()` will be visible to the thread that’s released from the associated `wait()`.

- Modifications made by a thread that terminates are visible to a thread that joins the terminated thread [must call `join()`].

- Memory initialized in a `static` initializer is safely accessible by all threads, including the one that caused the class-file load.
A Few Articles on SMP Problems

- Paul Jakubik (ObjectSpace)
  - www.primenet.com/~jakubik/mpsafe/MultiprocessorSafe.pdf

- Bill Pugh (Univ. of Maryland) mailing list
  - www.cs.umd.edu/~pugh/java/memoryModel/

- Allen Holub

- Brian Goetz
Memory-Model JSR

- JSR-000113: Memory Model and Thread Specification Revision
  - [http://www.javasoft.com/aboutJava/communityprocess/jsr/jsr_133.html](http://www.javasoft.com/aboutJava/communityprocess/jsr/jsr_133.html)

- But it’ll take time to implement, and may not be implemented correctly
Deadlock: The Simplest Scenario (1)

- Two or more threads, all waiting for each other
- Threads trying to acquire multiple locks, but in different order
Deadlock: The Simplest Scenario (2)

double field1; Object lock1 = new Object();
double field2; Object lock2 = new Object();
public void pebbles()
{   synchronized(lock1){ field1 = 0; }
}
public void bambam()
{   synchronized(lock2){ field2 = 0; }
}
public void fred()
{   synchronized(lock2)
{   synchronized(lock1)
{   field2 += field1;
}
}
}
public void wilma()
{   synchronized(lock1)
{   synchronized(lock2)
{   field2 -= field1;
}
}
Deadlock: The Simplest Scenario (3)

double field1; Object lock1 = new Object();
double field2; Object lock2 = new Object();
public void pebbles()
    {    synchronized(lock1){ field1 = 0; } }
public void bambam()
    {    synchronized(lock2){ field2 = 0; } }
public void fred()
    {    synchronized(lock2){
        synchronized(lock1)
            {    field2 += field1;
            }
    }
}
public void wilma()
    {    synchronized(lock1){
        synchronized(lock2)
            {    field2 -= field1;
            }
    }
}
Deadlock: A More-Realistic Scenario

class Boss
{
    private Sidekick robin;
    synchronized
    void set_side_kick(Sidekick kid)
    {
        robin = kid;
    }
    synchronized void to_the_bat_cave()
    {
        robin.lets_go();
    }
    synchronized void report(String s)
    { /*...*/}
}

class Sidekick
{
    private Boss batman;
    Sidekick(Boss boss)
    {
        batman = boss;
    }
    synchronized void lets_go()
    {
        batman.report("yeah boss");
    }
    synchronized void sock_bam()
    {
        batman.report("Ouch!");
    }
}

Boss  batman = new Boss();
Sidekick robin  = new Sidekick(batman);
batman.set_side_kick( robin );

1. Thread 1 (Alfred) calls batman.to_the_bat_cave(); Alfred now has the lock on batman
2. Thread 1 is preempted just before calling lets_go()
3. Thread 2 (Joker) calls robin.sock_bam()—Joker now has the lock on robin
4. Robin tries to report() to batman (on thread 2), but can't because Alfred has the lock. Joker is blocked
5. Thread 1 wakes up, tries to call lets_go(), but can't because Joker has the lock
Nested-monitor Lockout

- Can happen any time you call a method that can block from any synchronized method
- Consider the following (I’ve removed exception handling):

```java
class Black_hole
{
    private InputStream input =
        new Socket("www.holub.com",80)
            .getInputStream();

    public synchronized int read()
    {
        return input.read();
    }
    public synchronized void close()
    {
        input.close();
    }
}
```

How do you close the socket?
Nested-monitor Lockout: Another Example

- The notifying queue blocks if you try to dequeue from an empty queue

```java
class Black_hole2
{
    Notifying_queue queue =
        new Notifying_queue();

    public synchronized void put(Object thing)
    {
        queue.enqueue(thing);
    }

    public synchronized Object get()
    {
        return queue.dequeue();
    }
}
```
Why Was `stop()` Deprecated?

- NT leaves DLLs (including some system DLLs) in an unstable state when threads are stopped externally.

- `stop()` causes all monitors held by that thread to be released:
  - But thread may be stopped half way through modifying an object, and
  - Other threads can access the partially modified (now unlocked) object.
Why Was `stop()` Deprecated (2)?

- The only way to safely terminate a thread is for `run()` to return normally.
- Code written to depend on an external `stop()` will have to be rewritten to use `interrupted()` or `isInterrupted()`.
interrupt(), don’t stop()

class Wrong
{ private Thread t =
   new Thread()
   { public void run()
      { while( true )
         { //...
           blocking_call();
         }
      }
   };
   public stop()
   { t.stop();
   }
}

• But there’s no safe way to stop a thread that doesn’t check the “interrupted” flag

class Right
{ private Thread t =
   new Thread()
   { public void run()
      { try
         { while( !isInterrupted() )
            { //...
              blocking_call();
            }
         }
      }catch(InterruptedException e)
         {/*ignore, stop request*/}
      }
   };
   public stop()
   { t.interrupt();
   }
interrupt() gotchas

- `interrupt()` works only with the methods of the `Thread` and `Object` classes (e.g., `wait()`, `sleep()`, `join()`, etc.)

- It is not possible to interrupt out of a blocking I/O operation like `read()`
  - You can break out of a socket read by closing the socket, but that’s hideous
Why Were `suspend()` and `resume()` Deprecated?

- The `suspend()` method does not release the lock

```java
class Wrong {
    public synchronized void take_a_nap() {
        suspend();
    }
    public synchronized void wake_up() {
        resume();
    }
}
```

Once a thread has entered `take_a_nap()`, all other threads will block on a call to `wake_up()`. (Someone has gone into the bathroom, locked the door, and fallen into a drug-induced coma)

```java
class Right {
    public synchronized void take_a_nap() {
        try {
            wait();
        } catch (InterruptedException e) {
            /*do something reasonable*/
        }
        notify();
    }
}
```

The lock is released by `wait()` before the thread is suspended.
The Big-picture Coding Issues

- Design-to-coding ratio is 10:1 in threaded systems
- Formal code inspection or pair programming is essential
- Debugging multithreaded code takes longer
  - Bugs are usually timing related
- It’s not possible to fully debug multithreaded code in a visual debugger
  - Instrumented VMs cannot find all the problems because they change timing
  - Classic Heisenberg uncertainty: observing the process impacts the process
- Complexity can be reduced with architectural solutions (e.g., Active Objects)
Given That the Best Solution Isn’t Finding a New Profession…

- Low-level solutions (roll-your-own semaphores)
  - I’ll look at a few of the simpler classes covered in depth in *Taming Java Threads*
  - My intent is to give you a feel for multithreaded programming, not to provide an exhaustive toolkit

- Architectural solutions (active objects, etc.)
Roll Your Own (A Catalog)

• **Exclusion Semaphore** *(mutex)*
  – Only one thread can own at one time
  – Roll-your-own version can contain a timeout

• **Condition Variable**
  – Wait while condition false
  – Roll-your-own version can have state

• **Counting Semaphore**
  – Control pool of resources
  – Blocks if resource is unavailable
Roll Your Own (Cont.)

• **Message Queues** (interthread communication)
  – Thread blocks until a message is enqueued
  – Typically, only thread per queue

• **Thread Pools**
  – A group of dormant threads wait for something to do
  – A thread activates to perform an arbitrary task

• **Timers**
  – Allow operation to be performed at regular intervals
    • Block until a predetermined time interval has elapsed
    • Block until a predetermined time arrives
• **Reader/Writer Locks**
  - Allow thread-safe access to global resources such as files:
    • Must acquire the lock to access a resource
    • Writing threads are blocked while a read or write operation is in progress
    • Reading threads are blocked only while a write operation is in progress. Simultaneous reads are okay
Threads From an OO Perspective

• Think messages, not functions

• **Synchronous messages**—handler doesn’t return until it’s done doing whatever sender requests

• **Asynchronous messages**—handler returns immediately—Meanwhile request is processed in the background

```java
toolkit.getDefaultToolkit().getImage(some_URL);
```
The Java™ Programming Language Threading Model Is Not OO

- No language-level support for asynchronous messaging
- Threading system is based entirely on procedural notions of control flow
- Deriving from `Thread` is misleading
- Novice programmers think that all methods of a class that extends `Thread` run on that thread, when in reality, the only methods that run on a thread are methods that are called either directly or indirectly by `run()`
Implementing Asynchronous Methods—One-Thread-Per-Method

class Receiver
{
    //...
    public asynch_method()
    {
        new Thread()
        {
            public void run()
            {
                synchronized( Receiver.this )
                {
                    // Make local copies of
                    // outer-class fields here.
                }
                // Code here doesn't access outer
                // class (or uses only constants).
            }
            .start();
        }
    }
}
// This class demonstrates an asynchronous flush of a
// buffer to an arbitrary output stream

class Flush_example
{
  public interface Error_handler
  {
    void error( IOException e );
  }
  private final OutputStream out;
  private Reader_writer lock =
      new Reader_writer();
  private byte[] buffer;
  private int length;

  public Flush_example( OutputStream out )
  {
    this.out = out;
  }
}
synchronized void flush( final Error_handler handler )
{  new Thread()    // Outer object is locked
   { byte[] copy;  // while initializer runs.
   { copy = new byte[Flush_example.this.length];
      System.arraycopy(Flush_example.this.buffer,
                       0, copy, 0, Flush_example.this.length]);
      Flush_example.this.length = 0;
   }
   public void run()    // Lock is released
   { try    // when run executes
   {     lock.request_write();
      out.write( copy, 0, copy.length );
   }
   catch( IOException e ){ handler.error(e); }
   finally{ lock.write_accomplished(); }
   }
   .start();
}
A More Realistic One-Thread-Per-Method Strategy

• It is a worse-case synchronization scenario
  – Many threads all access the same outer-class object simultaneously
  – Since synchronization is required, all but one of the threads are typically blocked, waiting to access the object

• Thread-creation overhead can be stiff:
  
  Create String = .0040 ms.
  Create Thread = .0491 ms.
  Create & start Thread = .8021 ms. (NT 4.0, 600MHz)
Use Thread Pools

• The real version:
  – Grows from the initial size to a specified maximum if necessary
  – Shrinks back down to original size when extra threads aren’t needed
  – Supports a “lazy” close

```java
public final class Simplified_Thread_pool {
    private final Blocking_queue pool = new Blocking_queue();
}
private final class Pooled_thread extends Thread
{
    public void run()
    {
        synchronized( Simplified_thread_pool.this )
        {
            try
            {
                while( !isInterrupted() )
                {
                    ((Runnable)(
                        pool.dequeue() )).run();
                }
            }
            catch(InterruptedException  e){/* ignore */}
            catch(Blocking_queue.Closed e){/* ignore */}
        }
    }
}
Implementing a Simple Thread Pool

```java
public Simplified_Thread_pool(int pool_size )
{
    synchronized( this )
    {
        while( --pool_size >= 0 )
        {
            new Pooled_thread().start();
        }
    }
}
	public synchronized void execute(Runnable action) 
	{
	    pool.enqueue( action );
	}
	public synchronized void close()
	{
	    pool.close();
	}
```
The Active Object Design Pattern

- An architectural solution to threading synchronization

- Asynchronous requests are executed serially on a thread created for that purpose

- Think Tasks
  - An I/O task, for example, accepts asynchronous read requests to a single file and executes them serially
  - Message-oriented Middleware (MQS, Tibco …)
  - Ada and Intel RMX (circa 1979)
A Generalized Active Object

- The solution can be generalized in the Java programming language like this:

```java
run()
{
    while(true)
    {
        Runnable obj = queue.dequeue();
        obj.run();
    }
}

void message()
{
    queue.enqueue
    (  new Runnable()
        {  public void run()
            {  // do work here
            }
        }
    );
}
```
The javax.swing.* Thread Is an Active Object

- The Java™ Foundation Classes API (JFC/Swing)/AWT uses its own thread to handle the incoming OS-level messages and to dispatch appropriate notifications to listeners
- JFC/Swing is not thread safe
- The JFC/Swing subsystem is effectively a "UI task" to which you enqueue requests:

```java
SwingUtilities.invokeLater // enqueue a request
(new Runnable()
{
    public void run()
    {
        some_window.setSize(200,100);
    }

});
```
public class Active_object extends Thread
{
    private Msg_queue requests = new Msg_queue();
    public Active_object(){ setDaemon( true ); }
    public void run()
    {
        try
        {
            Runnable request;
            while((request=(Runnable)(
                  requests.dequeue()))!= null)
            {
                request.run();
                request = null;  yield();
            }
        }
        catch( InterruptedException e ){}
    }

    public final void dispatch(Runnable operation )
    {
        requests.enqueue( operation );
    }
}
Using an Active Object
(Detangling UNIX® Console Output)

class Console
{
    private static Active_object dispatcher
        = new Active_object();
    static { dispatcher.start(); }
    private Console() {}

    public static void println(final String s)
    {
        dispatcher.dispatch
            ( new Runnable()
                {
                    public void run()
                    {
                        System.out.println(s);
                    }
                }
            );
    }
}
Summing Up

- Java™ programming language threads ("Java threads") are not platform independent—they can’t be
- You have to worry about threads, like it or not
  - GUI code is multithreaded
  - No telling where your code will be used in the future
- Programming threads is neither easy nor intuitive
- Synchronized is your friend—Grit your teeth and use it
- Supplement language-level primitives to do real work
- The threading system isn’t object oriented
- Use good architecture, not semaphores
In-depth Coverage and Code

• For in-depth coverage, see Taming Java™ Threads
  www.apress.com

• For source code, etc., go to my web page
  www.holub.com