

# Multi-Threaded Concurrency in Java



---

Kimberly Bobrow Jennery

[kimberly.bobrow@sun.com](mailto:kimberly.bobrow@sun.com)

[kimberly@bobrow.net](mailto:kimberly@bobrow.net)

[kimberly@jennery.com](mailto:kimberly@jennery.com)

Sun Microsystems



# Overview of This Session

---

- What is a Thread?
- Thread States
- Thread Priorities
- Semaphore/signaling Concepts
- Creating Threads
- Daemon Threads
- Thread Groups
- Thread Methods



# Overview *(Continued)*

---

- Synchronization
- Avoiding Deadlocks, and miscellaneous tips
- Demos and Discussion



# Multi-Threading Examples

---

- Multiple file I/O / Database access
- Access to the network
- Communication programs
- Long computations
- Animation - screen updates
- Preliminary data entry validation
- Logically parallel tasks
- Anything the user or program doesn't HAVE to wait for



# Semaphore Concepts

---

- Critical Sections
- Mutual Exclusion
- Event Signaling
- Multiple Events



# Critical Sections

---

- The concept that one thread must own the processor for the duration of some section of code, to the exclusion of all other threads
- Deadlock possible or even likely if critical section waits for a resource
- A method of synchronization or signaling should be used for safety



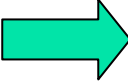
# Mutual Exclusion Semaphores

---

- One thread has control of/access to a resource (can run) or another does, but not both.
- For serializing access to some resource
  - file
  - memory
  - *etc.*
- Order can't matter, only that they don't run over each other

# Mutual Exclusion Semaphores

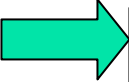
Thread A



```
// do something  
  
// request the  
// semaphore  
.  
. // use resource  
.  
// release the  
// semaphore  
  
// do something
```

running/runnable

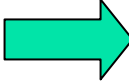
Thread B



```
// do something  
  
// request the  
// semaphore  
.  
. // use resource  
.  
// release the  
// semaphore  
  
// do something
```

running/runnable

Thread C

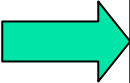


```
// do something  
  
// request the  
// semaphore  
.  
. // use resource  
.  
// release the  
// semaphore  
  
// do something
```

running/runnable

# Mutual Exclusion Semaphores

Thread A

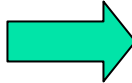


```
// do something
// request the
// semaphore
.
. // use resource
.
// release the
// semaphore

// do something
```

running/runnable

Thread B

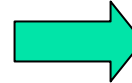


```
// do something
// request the
// semaphore
.
. // use resource
.
// release the
// semaphore

// do something
```

running/runnable

Thread C



```
// do something
// request the
// semaphore
.
. // use resource
.
// release the
// semaphore

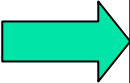
// do something
```

running/runnable

## Who gets the semaphore???

# Mutual Exclusion Semaphores

## Thread A

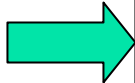


```
// do something
// request the
// semaphore
.
. // use resource
.
// release the
// semaphore

// do something
```

Not runnable

## Thread B

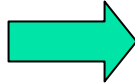


```
// do something
// request the
// semaphore
.
. // use resource
.
// release the
// semaphore

// do something
```

running/runnable

## Thread C



```
// do something
// request the
// semaphore
.
. // use resource
.
// release the
// semaphore

// do something
```

Not runnable

# Mutual Exclusion Semaphores

Thread A

```
// do something
// request the
// semaphore
.
. // use resource
.
// release the
// semaphore

// do something
```

Not runnable

Thread B

```
// do something
// request the
// semaphore
.
. // use resource
.
// release the
// semaphore

// do something
```

running/runnable

Thread C

```
// do something
// request the
// semaphore
.
. // use resource
.
// release the
// semaphore

// do something
```

Not runnable

## Who gets the semaphore???

# Mutual Exclusion Semaphores

## Thread A

```
// do something

// request the
// semaphore

.
. // use resource
.
// release the
// semaphore

// do something
```

Not runnable

## Thread B

```
// do something

// request the
// semaphore

.
. // use resource
.
// release the
// semaphore

// do something
```

running/runnable

## Thread C

```
// do something

// request the
// semaphore

.
. // use resource
.
// release the
// semaphore

// do something
```

Not runnable

**And so on...**



# Event Semaphores

---

- Used to signal one or more threads of an event
- Two analogies:
  - Horse race starting gate
  - Traffic light
- There is no "owner" of the semaphore.
- Any thread with a 'handle' may green light (post) or red light (reset) the semaphore at any time



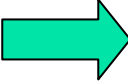
# Event Semaphores

---

- Horse race
  - All horses wait at the gate (set state) until the starting gun (event) goes off (posted), then they ALL go
- Traffic light
  - Turn the light green (post)
  - Turn the light red (set)
  - Wait – see if the light is green or red, proceed accordingly

# Event Semaphores

## Thread A



```
// create it
// 'red' (set)

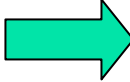
// wait for the
// semaphore

// event must
// have happened

// keep going
```

running/runnable

## Thread B



```
// do something

// wait for the
// semaphore

// event must
// have happened

// keep going
```

running/runnable

## Thread C

```
// doing
// something

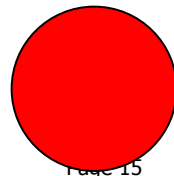
// event everyone
// is waiting on

// 'green light'
// the semaphore

// keep going
```

Not started yet

RED LIGHT (SET)



# Event Semaphores

## Thread A

```
// create it
// 'red' (set)

// wait for the
// semaphore

// event must
// have happened

// keep going
```

Not runnable

## Thread B

```
// do something

// wait for the
// semaphore

// event must
// have happened

// keep going
```

Not runnable

## Thread C

```
// doing
// something

// event everyone
// is waiting on

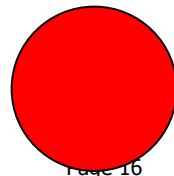
// 'green light'
// the semaphore

// keep going
```

running

**Thread A and B are waiting for the light to turn green (posted)**

RED LIGHT (SET)



# Event Semaphores

## Thread A

```
// create it
// 'red' (set)

// wait for the
// semaphore

// event must
// have happened

// keep going
```

Not runnable

## Thread B

```
// do something

// wait for the
// semaphore

// event must
// have happened

// keep going
```

Not runnable

## Thread C

```
// doing
// something

// event everyone
// is waiting on

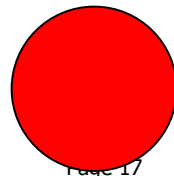
// 'green light'
// the semaphore

// keep going
```

running

**Thread A and B are waiting for the light to turn green (posted)**

RED LIGHT (SET)



# Event Semaphores

## Thread A

```
// create it
// 'red' (set)

// wait for the
// semaphore

// event must
// have happened

// keep going
```

Ready/running

## Thread B

```
// do something

// wait for the
// semaphore

// event must
// have happened

// keep going
```

Ready/running

## Thread C

```
// doing
// something

// event everyone
// is waiting on

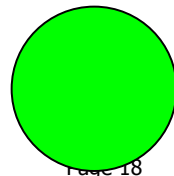
// 'green light'
// the semaphore

// keep going
```

Ready/running

**Thread A and B are unblocked  
and ready to go!**

GREEN LIGHT  
(POSTED)



# Event Semaphores

## Thread A

```
// create it
// 'red' (set)

// wait for the
// semaphore

// event must
// have happened

// keep going
```

Ready/running

## Thread B

```
// do something

// wait for the
// semaphore

// event must
// have happened

// keep going
```

Ready/running

## Thread C

```
// doing
// something

// event everyone
// is waiting on

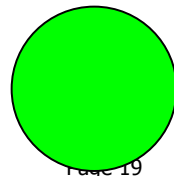
// 'green light'
// the semaphore

// keep going
```

Ready/running

**And so on...**

GREEN LIGHT  
(POSTED)





# Multiple Events

---

- The semaphore concept of waiting for a set of events, or resources
- Can wait on:
  - ANY of them or
  - ALL of them
  - Four combinations:
    - exclusive access to ALL resources in a collection
    - exclusive access to ANY resource in a collection
    - waiting for ALL events in a collection
    - waiting for ANY event in a collection

# Semaphore Wrap Up

- These concepts exist in any multi-threaded/multi-tasking environment
- Java 1.0-1.4 doesn't have a built in mechanism to specifically support each of these types of semaphores
- JSR-166 is currently scheduled to be incorporated into "Tiger," Java 1.5. JSR-166 has many new multi-threading features based on Doug Lea's concurrent classes
- Concurrency JSR-166 Interest site:  
<http://gee.cs.oswego.edu/dl/concurrency-interest/index.html>
- Doug Lea's existing concurrency classes can be used in the meantime instead of "rolling your own." They can be found at:

<http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html>



# Threads

---

- Are functions which run asynchronously (in the case of a multi-processor system) or “apparently asynchronously”
- Start at well-known pre-defined locations (`main()` or `run()`)
- Are executed by the JVM thread scheduler based on a variety of rules and available resources
- Are methods in separate threads, and thus have their own local variables
- Share static data between all threads in a Java program

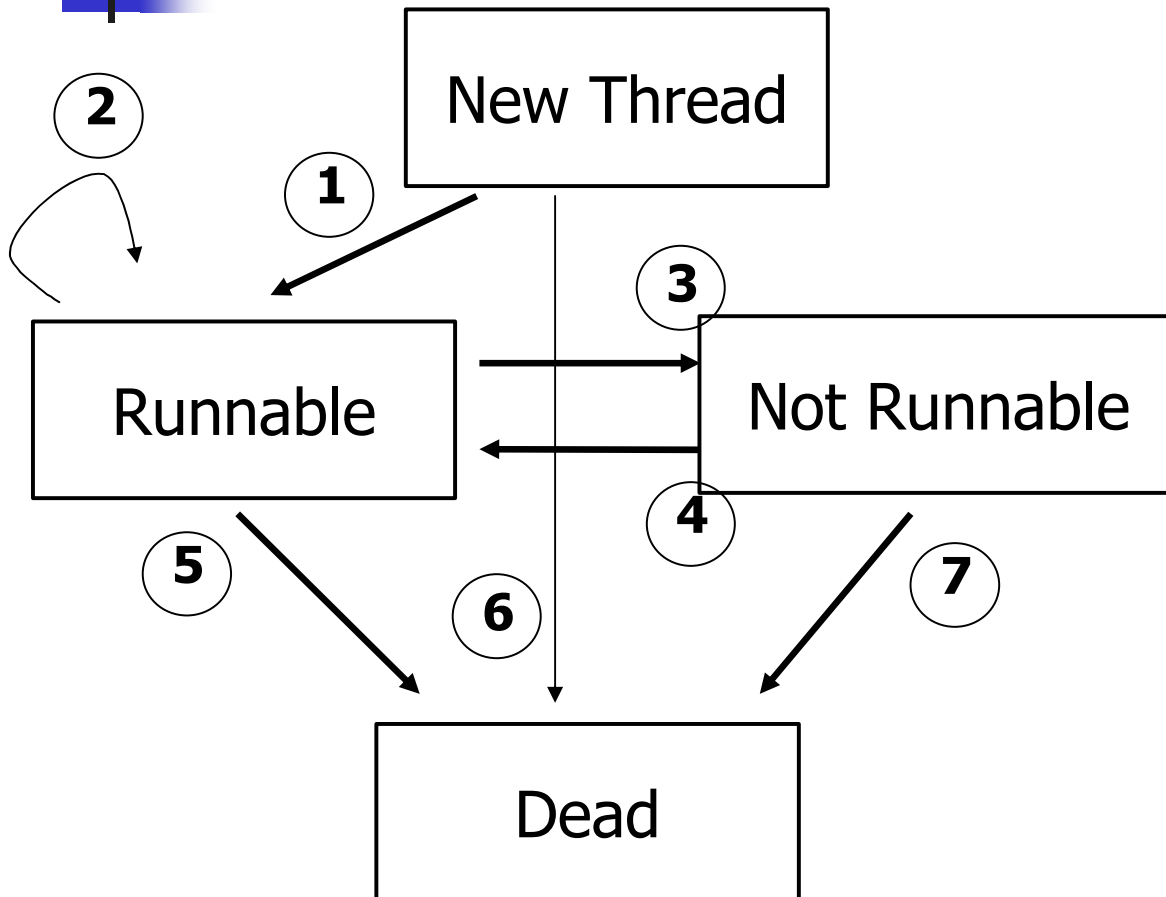


# Threads

---

- When the JVM starts up there is usually a single non-daemon thread which typically calls the `main` method of some designated class
- The JVM continues to execute threads until either of the following occurs:
  - The `exit` method of class `Runtime` has been called and the security manager has permitted the exit operation to take place.
  - All threads that are not daemon threads have died, either by returning from the call to the `run` method or by throwing an exception that propagates beyond the `run` method.

# Thread States



- 1) method invoked  
start
- 2) method invoked  
yield
- 3) Wait condition  
blocked i/o  
wait  
sleep, suspend, *etc.*
- 4) Wait condition satisfied  
notify, *etc.*
- 5) stop, or run exits
- 6) method invoked  
stop
- 7) method invoked  
stop



# Thread Priorities

---

- Thread scheduler decides which non-blocked thread to dispatch based on priority
- Pre-emptive scheduling
- Timeslicing within equal priorities is currently system dependent
- Don't write code which depends on time-sharing/slicing – use `yield()` in tight loops



# Thread Priorities

---

- Integer between:
  - `Thread.MIN_PRIORITY` (= 1)
  - `Thread.MAX_PRIORITY` (= 10)
- Default:
  - `Thread.NORM_PRIORITY` (= 5)
- Methods
  - `void setPriority(int p)`
  - `int getPriority()`



# Creating Threads

---

- Extend Thread class
  - Override run() method
  - Use if you simply want to run something asynchronously, but don't have a class with existing behavior you want to use
- Extend other class using Runnable interface
  - Override run() method
  - Create new Thread object, and pass Runnable object during Thread instantiation
  - Use if you have an existing class with methods you want to run asynchronously



# Extend Thread Class

---

```
class ThreadSample {
    public static void main(String[ ] args) {
        DummyThread d = new DummyThread();
        d.start();
        // do something main()-like here
    }
}

class DummyThread extends Thread {
    public void run() {
        // do something asynch here
    }
}
```

# Using Runnable Interface

```
class ThreadSample {
    public static void main(String[] args)
    {
        Thread d = new Thread(new DummyThread());
        d.start();
        // do something main()-like here
    }
}

class DummyThread extends DummyOther implements Runnable
{
    public void run()
    {
        // do something asynch here
    }
}
```



# Using Runnable Interface

```
class ThreadSample {
public static void main(String[] args)
    {
    Thread d = new Thread(new ThreadedCustomer());
    d.start();
    // do something main()-like here
    }
}

class ThreadedCustomer extends Customer implements Runnable {
public void run()
    {
    // do something asynch here
    }
}
```



# Important Considerations

---

- Never, EVER make assumptions about the order in which threads will execute
- If the order of processing is important between threads, use some mechanism of inter-thread communication to control it
- Debugging can be tricky – the debugger will almost certainly change the timing of threads



# Let's look at Some Sample Code

---

ThreadTest1

ThreadTest2

# Interrupting a Thread

- If you need to stop a thread which may be waiting for long periods you can interrupt it with `Thread.interrupt()`
- Any thread which catches an interrupt exception must do one of the following:
  - Deal with the situation immediately
  - Reassert the exception with:
    - `Thread.currentThread().interrupt();`
  - Rethrow the Exception, if allowed
- Note: Threads do not always respond to interruptions
- Check the interrupt state with:
  - `public static boolean interrupted()` – returns true/false results of current thread's interrupt state and resets the state
  - `public boolean isInterrupted()` – returns true/false results of current thread's interrupt state and leaves the status as is

# How to Kill a Thread, or: Cooperative Suicide

- Developers are not given a reliable, programmatic mechanism for killing a thread from the outside.
- `stop()` `suspend()` and `resume()` have all been deprecated since 1.2, for good reason.
  - `stop()` is inherently unsafe because you have no way to know what state objects running in the thread are in, if you stop it from the outside.
  - `suspend()` is likewise unsafe, with the added danger of the thread not releasing any locks it holds while suspended. Deadlocks are likely.
- If a thread is going to need to potentially be stopped from the outside, this needs to be prearranged and programmed for with a simple "suicide pact".



# Stopping a Thread – Method 1

---

- Follow these simple steps to allow your thread to be stopped from the outside:
  - create a volatile boolean flag variable for your Thread class (default false).
  - create a method which sets the flag to true when called (clobber() or whatever).
  - periodically, within your code, check the value of the flag and if it's true, immediately (and gracefully) get out.



# Stopping a Thread

It has always been recommended that you terminate threads through thread cooperation - `stop()/suspend()` deprecation helps enforce this idea.

```
class test extends Thread {
    private volatile boolean stop;

    public void stopThread() {
        stop = true;
    }

    public void run() {
        stop = false;
        while (!stop) {
            // do work
        }
        // do cleanup
    }
}
```

# Stopping a Thread – Method 2

- Follow these simple steps to allow your thread to be stopped from the outside:
  - Use the interrupt status as your flag
  - Create a method which interrupts the thread when called (clobber() or whatever).
  - Periodically, within your code, check the interrupted state and if it's true, immediately (and gracefully) get out.
  - This is additionally useful if you need to kill a thread which spends time in wait() or sleep() – you will need to handle InterruptedException accordingly



# Daemon Threads

---

- Service provider threads
- Use `setDaemon(true)` to specify
- Use `isDaemon()` to determine if a thread is a Daemon thread
- When only daemon threads remain in a process, interpreter exits
- Often used for resource pools – when there are no more clients, the resource pool is no longer necessary



# Let's Look at Some Sample Code

---

- `DaemonThreadTest`



# Thread Groups

---

- For managing multiple threads as a group
- ThreadGroup class has methods for:
  - Collection Management
  - Operating on a Group
  - Operating on all Threads within a Group
  - Access Restriction to threads
- Possibly most useful for the typical developer is the ability for a ThreadGroup to catch all exceptions thrown by its Threads by overriding `uncaughtException(Thread, Throwable)`



# Thread Groups

---

- All threads belong to a ThreadGroup
  - If not specified, new threads are put in the default ThreadGroup
  - main ThreadGroup created at application start up
  - Threads are put in same group as thread that created it
- Threads cannot be moved to a new group after creation



# Creating a Thread in a Group

## Thread Constructors

```
public Thread(ThreadGroup group, Runnable target)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable target,
String name)
```

## Creating a thread as a member of a specified group

```
ThreadGroup dummyThreadGroup = new ThreadGroup("Some
Fine Group");
Thread myThread = new Thread(dummyThreadGroup, "Some
silly thread");
```



# Let's Look at Some Sample Code

---

- ThreadGroupTest1
- ThreadGroupTest2



# Synchronization

---

- Care needs to be taken to have not more than one thread access the same data at the same time
- Java offers implicit monitors (mutexes) as exclusive locks on objects by use of the keyword `synchronized`
- A method, or a block of code, may be protected by synchronization
- Synchronization is accomplished by using the keyword `'synchronized'`
- Only one synchronized method may run on any *object* at any one time



# synchronized keyword

```
class threadSample
{
    public synchronized void someMethod() {
        // code here is protected
    }

    public void someOtherMethod() {
        // an example of a synchronized block instead of method
        // some code here
        synchronized (someObj)
        {
            // some code for which it was important that someObj
            // was locked
        }
    }
}
```



# synchronized keyword

---

- Think of the word 'synchronized' to mean "request a specific lock" and that anyone requesting the same exact lock will be forced to wait until they obtain it before executing the code

# Every Object Has a Lock

Class:

Employee

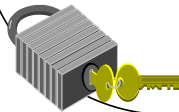
Objects:

Bob

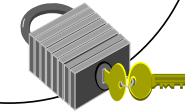
Sue

Elmer

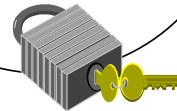
Employee:  
Bob Jones



Employee:  
Sue Smith



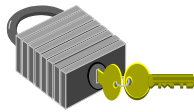
Employee:  
Elmer Fudd





---

**Employee:  
Bob Jones**

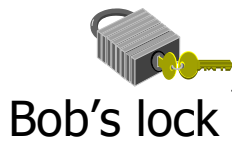


```
synchronized float getSalary ()  
{  
    // the salary can't be  
    // changed or set by other  
    // thread while we  
    // are in here - we must  
    // have the lock  
}
```

```
synchronized setSalary (float a)  
{  
    // the salary can't be  
    // read (or set) by any other  
    // thread while we  
    // are in here - we must  
    // have the lock  
}
```

Thread 1

bob.setSalary();

Thread2

elmer.setSalary();

Thread 3

bob.setSalary();



Waiting for  
Bob's lock

```
synchronized float getSalary ()
{
// the salary can't be
// changed or set by other
// thread while we
// are in here - we must
// have the lock
}
```

```
synchronized setSalary (float a)
{
// the salary can't be
// read (or set)by any other
// thread while we
// are in here - we must
// have the lock
}
```

# Java's Guarantees for *synchronized* Blocks

---

- One, and only one, thread can own a specific monitor/lock at any one time
- If a block of code, protected by the keyword 'synchronized' is executing, it **MUST** therefore own the associated lock

# Further *synchronized* Discussion

- The idea of synchronization in Java is both one of the most important, and one of the most misunderstood in its implementation
- Common misconceptions include:
  - a synchronized chunk of code is 'locked' and can never be running at the same time as another synchronized block.
  - the granularity of the requested lock is often not well understood.
  - not realizing that synchronization is on a per OBJECT basis.
- PLEASE ask questions – there is time allowed for it right now.

# Communication/ Synchronization

- Inter-thread communication is sometimes necessary in multi-threaded programs
- In order to avoid polling/looping practices, Java offers event waiting and notification using:
  - `wait()` – waits for a condition to be satisfied
  - `notify()` – notifies one waiting thread that an event has occurred
  - `notifyAll()` – notifies all waiting threads that an event has occurred
- Can only be called from synchronized methods or blocks – *i.e.* the lock must be obtained before making any of these calls



# Event Notification

## Thread 1

```
synchronized (x)
{
    // some before stuff
    x.wait();
    // some after stuff
}
```

## Thread 2

```
synchronized (x)
{
    // some before stuff
    x.wait();
    // some after stuff
}
```

## Thread 3

```
synchronized (x)
{
    // some before stuff
    x.notify();
    // some after stuff
}
```

- Java requires `wait()` and `notify()` to be called only from within synchronized blocks. *i.e.* you must own the monitor before waiting for the event.
- Does this make an impossible dead-lock situation? How can `notify()` in Thread 3 take place if the monitor is already owned by Thread 1 or 2?



# Event Notification

## Thread 1

```
synchronized (x)
{ // lock obtained
  // some before stuff
  x.wait();
  // some after stuff
} // lock released
```

## Thread 2

```
synchronized (x)
{ // lock obtained
  // some before stuff
  x.wait();
  // some after stuff
} // lock released
```

## Thread 3

```
synchronized (x)
{ // lock obtained
  // some before stuff
  x.notify();
  // some after stuff
} // lock released
```

- wait() releases the lock until a notify() condition is recognized
- The thread won't be able to continue until it has reobtained the lock (it must first be released by the current owner)

# wait() – Temporarily Releases a Lock within a synch. block

```
public void someMethod() {
    synchronized (someObj) {
        // we have the lock if we're here
        someObj.wait(); // this releases the lock,
                        // and blocks until notify()/notifyAll()
        someCode();    // the lock has been reacquired
                        // before this line will execute
    }
}
```



# Let's Look at Some Sample Code

---

- ThreadWaitTest1
- ThreadWaitTest2



# Thread Methods of Interest

---

- `currentThread()`
  - Returns a reference to the currently executing thread object.
- `dumpStack()`
  - Prints a stack trace of the current thread. Can be useful for debugging.
- `enumerate(Thread[])`
  - Copies into the specified array every active thread in this thread group and its subgroups.
- `getName()`
  - Returns this thread's name.
- `getPriority()`
  - Returns this thread's priority.



# Thread Methods of Interest

---

- `holdsLock(Object obj)` (1.4 only)
  - Returns true if and only if the current thread holds the monitor lock on the specified object.
- `interrupt()`
  - Interrupts this thread.
- `interrupted()`
  - Tests if the current thread has been interrupted. Clears the interrupt status.
- `isInterrupted()`
  - Tests if the current thread has been interrupted. Interrupt status unaffected.



# Thread Methods of Interest

---

- `join()`
  - Waits for this thread to die.
- `join(long millis)`
  - Waits at most *millis* milliseconds for this thread to die.
- `join(long millis, int nanos)`
  - Waits at most *millis* milliseconds plus *nanos* nanoseconds for this thread to die.
- `run()`
  - If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.



# Thread Methods of Interest

---

- `sleep(long millis)`
  - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- `sleep(long millis, int nanos)`
  - Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds.
- `start()`
  - Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.
- `toString()`
  - Returns a string representation of this thread, including the thread's name, priority, and thread group.
- `yield()`
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute.



# Avoiding Deadlocks

---

- Difficult to detect – good offense is the best defense
- Acquire multiple locks in the same order
- Know your code well



# Performance Issues and Tips

---

- Thread where applicable, but don't start dozens of threads if not needed – use thread pools for intensely multithreaded programs if possible.
- For operations involving multiple objects with state that must be synchronized for all of them, lock all objects
- Never depend on thread prioritization for exclusive access to an object
- Never depend solely on thread pre-emption or prioritization for performance



# Performance Issues and Tips

---

- If you are creating an object solely for the purpose of using it as a lock to synchronize on, the cheapest object is a zero element byte array
  - `byte[] lock = new byte[0];` // much cheaper than
  - `Object lock = new Object();` // this
- Vector and Hashtable are both fully synchronized – don't use them for a single-threaded program – you'll pay the performance penalty for no purpose.



# Performance Issues and Tips

---

- In general your data members should be private anyway, but especially if more than one thread can be accessing an object:
  - Make data members private
  - Use synchronized accessors
- In summary – any data that is protected somewhere in your code in a synchronized method should also be private to the class to be fully protected

# Miscellaneous Tidbits and Gotchas

- Local copies of variables in threads
  - The JVM may make local copies of global variables in each thread. Values are not guaranteed to be written to main memory unless the variable is either: a) updated within a synchronized block; or b) declared volatile
- Never reassign the object reference of a locked object!

```
synchronized (obj) {  
    // DON'T do this:  
    obj = new Object();  
    // I don't have the  
    // lock on obj  
}
```

```
synchronized (obj) {  
    // I may not be  
    // using obj safely  
}
```

# Miscellaneous Tidbits and Gotchas

- A thread may acquire the same lock multiple times, and will not release it until the count has gone back down to 0
- `Thread.sleep(5000)` is supposed to sleep for 5 seconds. However, if somebody changes the system time, you may sleep for a very long time or no time at all. The OS records the wake up time in absolute form, not relative.
- Thread scheduling is not guaranteed to be round-robin. A task may totally hog the CPU at the expense of threads of the same priority. You can use `Thread.yield()` to have a conscience.



# Threads – What Cost?

---

- Small CPU overhead at initialization
- Complicates design
- Requires parallel thinking
- Debugging can be very tricky
  - Running in a debugger can alter timing, making bugs harder to trace
  - Breakpoints in a thread does alter timing
- Adding threads to a project, if it wasn't designed with threads in mind can be particularly problematic



# Other Sessions To See

---

- Peter Hagggar's "Advanced Java Multi-threading techniques"
- Note especially his comments about atomicity, and wait()/notifyAll() processing – these are applicable in any multi-threading application, not just 'advanced' situations.



# References

---

*Java Threads*, Scott Oaks & Henry Wong, O'Reilly, ISBN 1-56592-216-6

*Java Tutorial*, Mary Campione & Kathy Walrath, Addison Wesley, ISBN 0-201-63454-6

*Java Handbook*, Patrick Naughton, McGraw Hill, ISBN 0-07-882199-1

*Practical Java by Peter Haggart*, Addison Wesley, ISBN 0201616467

*Concurrent Programming in Java*, Doug Lea, Addison Wesley ISBN 0201310090

*Thinking in Java*, Bruce Eckel, Prentice Hall, ISBN 0-13-659723-8

*The Java Language Specification*, Second Edition

# Demos, Discussion, Questions?

---

- (and if there's time – a relevant story with a moral)



# Contact Info

---

Kimberly Bobrow Jennery

[kimberly.bobrow@sun.com](mailto:kimberly.bobrow@sun.com)

[kimberly@jennery.com](mailto:kimberly@jennery.com)

[kimberly@bobrow.net](mailto:kimberly@bobrow.net)

[geekstress@javasluts.com](mailto:geekstress@javasluts.com) (don't ask)