CS 580 Client-Server Programming
Spring Semester, 2005
Doc 10 Threads part 2
Contents

References

Cancellable Activities, Doug Lea, October 1998, http://gee.cs.oswego.edu/dl/cpj/cancel.html

Concurrent Programming in Java: Design Principles and Patterns, Doug Lea, Addison-Wesley, 1997

The Java Programming Language, 2nd Ed. Arnold & Gosling, Addison-Wesley, 1998

Java's Atomic Assignment, Art Jolin, *Java Report*, August 1998, pp 27-36.

Java 1.4.2 on-line documentation http://java.sun.com/j2se/1.4.2/docs/api/overview-summary.html

Java Network Programming 2nd Ed., Harold, O'Reilly, Chapter 5

Reading

Java Network Programming, 3nd Ed., Harold, Chapter 5. (Java)

Thread Control
Java interrupt ()

Sent to a thread to interrupt it

A thread has interrupted status flag

JDK 1.4 Doc state

InterruptedException is thrown if thread is blocked a call to:
- wait
- join
- sleep

and the interrupted status flag is cleared

ClosedByInterruptException is thrown if the thread is blocked
- I/O operation on an interruptible channel

and the interrupted status flag is set

Interruptible channels are part of JDK 1.4 NIO package

If the thread is blocked by a selector:
- Interrupt status is set
- The thread returns from the selector call as normal

If none of the other conditions hold then the thread's interrupt status is set

Interrupt and Pre JDK 1.4 NIO operations

If a thread is blocked on a read/write to a:

- Stream
- Reader/Writer
- Pre-JDK 1.4 style socket read/write

The interrupt does not interrupt the read/write operation!

The threads interrupt flag is set

Until the IO is complete the interrupt has no effect

This is one motivation for the NIO package

Interrupt does not stop a Thread

The following program does not end
The interrupt just sets the interrupt flag!

```
public class NoInterruptThread extends Thread {
   public void run() {
      while ( true) {
         System.out.println(  "From: " + getName()  );
      }
   }

   public static void main(String args[]) throws InterruptedException{

      NoInterruptThread focused  = new NoInterruptThread( );
      focused.setPriority( 2 );
      focused.start();
      Thread.currentThread().sleep( 5 ); // Let other thread run
      focused.interrupt();
      System.out.println( "End of main");
   }
}
```

Output

```
From: Thread-0 (repeated many times)
End of main
From: Thread-0 (repeated until program is killed)
```

Using Thread.interrupted

This example uses the test Thread.interrupted() to allow the thread to be continue execution later.

```
public class RepeatableNiceThread extends Thread {
   public void run() {

      while ( true ) {
         while ( !Thread.interrupted() )
            System.out.println(  "From: " + getName()  );

         System.out.println( "Clean up operations" );
      }
   }

   public static void main(String args[]) throws InterruptedException{

      RepeatableNiceThread  missManners  =
            new RepeatableNiceThread( );
      missManners.setPriority( 2 );
      missManners.start();

      Thread.currentThread().sleep( 5 );
      missManners.interrupt();
      }
}
```

Output

```
From: Thread-0
Clean up operations
From: Thread-0
From: Thread-0 (repeated)
```

Interrupt and sleep, join & wait

Let thread A be in the not runnable state due to being sent either the sleep(), join() or wait() methods. Then if thread A is sent the interrupt() method, it is moved to the runnable state and InterruptedException is raised in thread A.

In the example below, NiceThread puts itself to sleep. While asleep it is sent the interrupt() method. The code then executes the catch block.

```
public class  NiceThread  extends Thread {
   public void run() {
      try {
         System.out.println( "Thread started");
         while ( !isInterrupted() ) {
            sleep( 5 );
            System.out.println(  "From: " + getName()  );
         }
         System.out.println( "Clean up operations" );
      } catch ( InterruptedException interrupted ) {
         System.out.println( "In catch" );
      }
   }

   public static void main( String args[] )  {
      NiceThread  missManners  = new NiceThread( );
      missManners.setPriority( 6 );
      missManners.start();
      missManners.interrupt();
   }
}
```

Output

```
Thread started
From: Thread-0
From: Thread-0
In catch
```

Who Sends sleep() is Important

Since main sends the sleep method, not the thread itself, the InterruptedException is not thrown.

```
public class WhoSendsSleep extends Thread {
   public void run() {
      try {
         while ( !isInterrupted() ) {
            System.out.println(  "From: " + getName()  );
         }
         System.out.println( "Clean up operations" );
      } catch ( Exception interrupted ) {
         System.out.println( "In catch" );
      }
   }

   public static void main( String args[] ) {
      try {
         NiceThread  missManners  = new NiceThread( );
         missManners.setPriority( 1 );
         missManners.start();
         missManners.sleep( 50);   //Which thread is sleeping?
         missManners.interrupt();
      } catch ( InterruptedException interrupted ) {
         System.out.println( "Caught napping" );
      }
   }
}
```

Output

```
Thread started
From: Thread-0
From: Thread-0
Clean up operations
```

Threads & Method Sends

A method is executed in the thread that sends the method

```
missManners.sleep( 50);
```

Put the current thread to sleep not the missManners thread

Safety - Mutual Access

With multiprocessing we need to address mutual access by different threads. When two or more threads simultaneously access the same data there may be problems.

Some types of access are safe. If a method accesses just local data, then multiple threads can safely call the method on the same object. Assignment statements of all types, except long and double, are atomic. That is a thread can not be interrupted by another thread while performing an atomic operation.

Java Safety - Synchronize

Synchronize is Java's mechanism to insure that only one thread at a time will access a piece of code. We can synchronize methods and block's of code (synchronize statements).

Synchronized Instance Methods

When a thread executes a synchronized instance method on an object, that object is locked. The object is locked until the method ends. No other thread can execute any synchronized instance method on that object until the lock is released. The thread that has the lock can execute multiple synchronized methods on the same object. The synchronization is on a per object bases. If you have two objects, then different threads can simultaneously execute synchronized methods on different objects. Unsynchronized methods can be executed on a locked object by any thread at any time. The JVM insures that only one thread can obtain a lock on an object at a time.

```
class SynchronizeExample {
   int[]  data;

   public String toString() {
      return "array length " + data.length + " array values " + data[0];
   }

   public synchronized void initialize( int size,  int  startValue){
      data  =  new  int[ size ];
      for  ( int  index  = 0; index < size;  index++ )
         data[ index ]  =  (int ) Math.sin( index * startValue );
   }

   public  void unSafeSetValue( int  newValue) {
      for  ( int  index  = 0; index < data.length;  index++ )
         data[ index ]  =  (int ) Math.sin( index * newValue );
   }

   public  synchronized void safeSetValue( int  newValue) {
      for  ( int  index  = 0; index < data.length;  index++ )
         data[ index ]  =  (int ) Math.sin( index * newValue );
   }
}
```

Synchronized Static Methods

A synchronized static method creates a lock on the class, not the object. When one thread has a lock on the class, no other thread can execute any synchronized static method of that class. Other threads can execute synchronized instance methods on objects of that class.

```
class SynchronizeStaticExample {
   int[]  data;
   static int[] classData;

   public synchronized void initialize( int size,  int  startValue){
      data  =  new  int[ size ];
      for  ( int  index  = 0; index < size;  index++ )
         data[ index ]  =  (int ) Math.sin( index * startValue );
   }

   public synchronized void initializeStatic( int size,  int  startValue){
      classData  =  new  int[ size ];
      for  ( int  index  = 0; index < size;  index++ )
         classData[ index ]  =  (int ) Math.sin( index * startValue );
   }
}
```

Synchronized Statements

A block of code can be synchronized. The basic syntax is:

```
synchronized ( expr ) {
   statements
}
```

The expr must evaluate to an object. This will lock the object. The lock is released when the thread finishes the block. Until the lock is released, no other thread can enter any method or synchronized block that is locked by the given object.

A synchronized method is syntactic sugar for a synchronized block.

```
class LockTest {
   public synchronized void enter() {
      System.out.println( "In enter");
   }
}
```

Is the same as:

```
class LockTest {
   public void enter() {
      synchronized ( this ) {
         System.out.println( "In enter");
      }
   }
}
```

Lock for Block and Method

This example shows that a lock on an object also locks all access to the object via synchronized methods.

```
public class LockExample extends Thread {
   private Lock myLock;

   public LockExample( Lock aLock ) {
      myLock = aLock;
   }
   public void run() {
      System.out.println( "Start run");
      myLock.enter();
      System.out.println( "End run");
   }
   public static void main( String args[] ) throws Exception {
      Lock aLock = new Lock();
      LockExample tester = new LockExample( aLock );

      synchronized ( aLock ) {
         System.out.println( "In Block");
         tester.start();
         System.out.println( "Before sleep");
         Thread.currentThread().sleep( 5000);
         System.out.println( "End Block");
      }
   }
}

class Lock {
   public synchronized void enter() {
      System.out.println( "In enter");
   }
}
```

Output

```
In Block
Start run
Before sleep
End Block
In enter
End run     (why does this come at the end?)
```

Synchronized and Inheritance

If you want a method in a subclass to be synchronized you must declare it to be synchronized.

```
class Top
    {
    public void synchronized left()
        {
        // do stuff
        }

    public void synchronized right()
        {
        // do stuff
        }
    }

class Bottom extends Top
    {
    public void left()
        {
        // not synchronized
        }

    public void right()
        {
        // do stuff not synchronized
        super.right();   // synchronized here
        // do stuff not synchronized
        }
```

wait and notify Methods in Object

wait and notify are some of the most useful thread operations.

public final void **wait**(timeout) throws InterruptedException

Causes a thread to wait until it is notified or the specified timeout expires.

Parameters:
timeout - the maximum time to wait in milliseconds

Throws: IllegalMonitorStateException
If the current thread is not the owner of the Object's monitor.

Throws: InterruptedException
Another thread has interrupted this thread.

public final void **wait**(timeout, nanos) throws InterruptedException
public final void **wait**() throws InterruptedException

public final void **notify**()
public final void **notifyAll**()

Notifies all of the threads waiting for a condition to change. Threads that are waiting are generally waiting for another thread to change some condition. Thus, the thread effecting a change that more than one thread is waiting for notifies all the waiting threads using the method notifyAll(). Threads that want to wait for a condition to change before proceeding can call wait(). The method notifyAll() can only be called from within a synchronized method.

wait - How to use

The thread waiting for a condition should look like:

```
synchronized void waitingMethod()
   {
   while ( ! condition )
      wait();

   Now do what you need to do when condition is true
   }
```

Note

Everything is executed in a synchronized method

The test condition is in loop not in an if statement

The wait suspends the thread it atomically releases the lock on the object

notify - How to Use

```
synchronized void changeMethod()
    {
    Change some value used in a condition test

    notify();
    }
```

wait and notify Example

Over the next five slides is a typical consumer-producer example. Producers "make" items, which they put into a queue. Consumers remove items from the queue. What happens when the consumer wishes to remove when the queue is empty? Using threads, we can have the consumer thread wait until a producer thread adds items to the queue.

```
import java.util.ArrayList;

public class SharedQueue {
   ArrayList elements = new ArrayList();
   public synchronized void append( Object item ) {
      elements.add( item);
      notify();
   }

   public synchronized Object get( ) {
      try {
         while ( elements.isEmpty() )
            wait();
      }
      catch (InterruptedException threadIsDone ) {
         return null;
      }
      return elements.remove( 0);
   }
}
```

wait and notify - Producer

```
public class Producer extends Thread
   {
   SharedQueue factory;
   int workSpeed;

   public Producer( String name, SharedQueue output, int speed )
      {
      setName(name);
      factory = output;
      workSpeed = speed;
      }

   public void run()
      {
      try
         {
         int product = 0;
         while (true) // work forever
            {
            System.out.println( getName() + " produced " + product);
            factory.append( getName() + String.valueOf( product) );
            product++;
            sleep( workSpeed);
            }
         }
      catch ( InterruptedException WorkedToDeath )
         {
         return;
         }
      }
   }
```

wait and notify - Consumer

```
class Consumer extends Thread
   {
   Queue localMall;
   int sleepDuration;

   public Consumer( String name, Queue input, int speed )
      {
      setName(name);
      localMall = input;
      sleepDuration = speed;
      }

   public void run()
      {
      try
         {
         while (true) // Shop until you drop
            {
            System.out.println( getName() + " got " +
                                 localMall.get());
            sleep( sleepDuration );
            }
         }
      catch ( InterruptedException endOfCreditCard )
         {
         return;
         }
      }
   }
```

wait and notify - Driver Program

```
public class ProducerConsumerExample
   {
   public  static  void  main( String  args[] ) throws Exception
      {
      SharedQueue wallmart = new SharedQueue();
      Producer nike = new Producer( "Nike", wallmart, 500 );
      Producer honda = new Producer( "Honda", wallmart, 1200 );
      Consumer valleyGirl = new Consumer( "Sue", wallmart, 400);
      Consumer valleyBoy = new Consumer( "Bob", wallmart, 900);
      Consumer dink = new Consumer( "Sam", wallmart, 2200);
      nike.start();
      honda.start();
      valleyGirl.start();
      valleyBoy.start();
      dink.start();
      }
   }
```

Output

Nike produced 0	Sue got Nike3	Honda produced 3
Honda produced 0	Nike produced 4	Bob got Honda3
Sue got Nike0	Sue got Nike4	Nike produced 8
Bob got Honda0	Honda produced	Sue got Nike8
Nike produced 1	Bob got Honda2	Nike produced 9
Sam got Nike1	Nike produced 5	Sue got Nike9
Nike produced 2	Sue got Nike5	Honda produced 4
Sue got Nike2	Nike produced 6	Bob got Honda4
Honda produced 1	Sam got Nike6	Nike produced 10
Bob got Honda1	Nike produced 7	Sue got Nike10
Nike produced 3	Sue got Nike7	Nike produced 11

Some Thread Issues & Ideas
Passing Data – Multiple Thread Access

Situation

An object is passed between threads

Issue

If multiple threads have a reference to the same object

When one thread changes the object the change is global

Example

```
anObject = anotherThreadObject.getFoo();   // line A
System.out.println( anObject);             // line B
```

If multiple threads have access to anObject

The state of anObject can change after line A ends and before line B starts!

This can cause debugging nightmares

Passing Data – Multiple Thread Access

Possible Solutions

Pass copies

Returning data

```
pubic foo getFoo() {
   return foo.clone();
}

foo
   ^foo copy
```

Parameters

```
anObject.doSomeMunging( bar.clone());

anObject doSomeMunging: bar copy
```

Passing Data – Multiple Thread Access
Possible Solutions

Immutable Objects

Pass objects that cannot change

Java's base type and Strings are immutable

Background Operations
Situation

Perform some operation in the background
At same time perform some operations in the foreground
Need to get the result when operation is done

Issue

Don't make the code sequential

Avoid polling

```
public class Poll  {
   public static void main( String args[] ) {

   TimeConsumingOperation background =
      new TimeConsumingOperation();
   background.start();

   while ( !background.isDone() ) {
      performSomethingElse;
   }

   Object neededInfo = background.getResult();
   }
}
```

Futures

A future starts a computation in a thread
When you need the result ask the future
You will block if the result is not ready

Smalltalk

Promise class in VisualWorks

```
| delayedAnswer realAnswer |
delayedAnswer := [aClient perform: 'computePi' ] promise.
Do some other work here
realAnswer := delayedAnswer value
```

Sample Java Future

```
class FutureWrapper {
   TimeConsumingOperation myOperation;

   public FutureWrapper() {
      myOperation = new TimeConsumingOperation();
      myOperation.start();
   }

   public Object value() {
      try {
         myOperation.join();
         return myOperation.getResult();
      } catch (InterruptedException trouble ) {
         DoWhatIsCorrectForYourApplication;
      }
   }
}

public class FutureExample  {
   public static void main( String args[] ) {

   FutureWrapper myWorker = new FutureWrapper();

   DoSomeStuff;
   DoMoreStuff;

   x = myWorker.value();
   }
}
```

Callbacks

Have the background thread call a method when it is done

Java Outline

```
class MasterThread {
   public void normalCallback( Object result ) {
      processResult;
   }

   public void someMethod() {
      compute;
      TimeConsumingOperation backGround =
         new TimeConsumingOperation( this );

      backGround.start();
      moreComputation;
   }
}

class TimeConsumingOperation extends Thread {
   MasterThread master;

   public TimeConsumingOperation( MasterThread aMaster ) {
      master = aMaster;
   }

   public void run() {
         DownLoadSomeData;
         PerformSomeComplexStuff;
         master.normalCallback( resultOfMyWork );
      }
   }
}
```