CS 580 Client-Server Programming
Fall Semester, 2000
Doc 9 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

The Java Language Specification, Gosling, Joy, Steele, Addison-Wesley, 1996, Chapter 17 Threads and Locks.

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

Java 1.2 on-line documentation http://java.sun.com/products/jdk/1.2/docs/index.html

Java 1.2 Thread Docs http://java.sun.com/products/jdk/1.2/docs/tooldocs/solaris/threads.html

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

Ending Thread Execution

A thread ends when its run method ends. At times program needs to permanently stop a running thread. For example, when a user uses a browser to access a web page. The browser will open a new network connection in a separate thread. If the user then cancels the request, the browser may need to "kill" the thread doing the down load.

Suspend, Resume, Stop, destroy

Java has some thread methods that will stop threads. However, these methods are not safe! They could leave your program in an inconsistent state or cause deadlock. Suspend, resume and stop do exist, but are not safe. They are deprecated in JDK 1.2. Destroy, while listed in the on-line API, was never implemented. It throws a NoSuchMethodError() in JDK 1.2.

These methods are deprecated in JDK 1.2 because they are not thread safe

What replaces Stop?

It turns out that there is no safe way to implement a method that will stop a thread in general. Doug Lea recommends a multiphase approach. First, use interrupt. If that fails, try starving the thread. If that also fails, giving the thread the minimum priority to reduce its impact. If these all fail and the situation calls for drastic action, then one can use stop(), perform clean up operations, then exit the program.

Forcing a Thread to Quit - Using interrupt()

A thread can perform a block of operations then check to see if it is interrupted. If it has been interrupted, then it can take "proper" action. Sometimes proper action is to clean up, then quit. Sometimes proper action may be to "reset itself" to be available to run again later. In the example below the sleep() method is throwing the InterruptedException.

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

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

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

      NiceThread  missManners  = new NiceThread( );
      missManners.setPriority( 2 );
      missManners.start();
      Thread.currentThread().sleep( 5 ); // Let other thread run
      missManners.interrupt();
   }
}
```

Output

```
From: Thread-3
From: Thread-3
From: Thread-3
From: Thread-3
From: Thread-3
Clean up operations
```

Interrupt Methods in java.lang.Thread

void interrupt()

Sent to a thread to interrupt it

boolean isInterrupted()

Sent to a thread to see if it has been sent the interrupt() method

Returns true if the thread has been sent the interrupt() method

static boolean interrupted()

Sent to the current thread to see if it has been sent the interrupt() method

Returns true if the thread has been sent the interrupt() method

Clears the interrupt flag in the current thread

Using Thread.interrupted

This example uses the test Thread.interrupted() to allow the thread to be continue execution later. Note that thread uses suspend() after it has made sure that all data is safe. This is harder to do in real life than the simple example here indicates. Using wait would be better here, but we have not covered wait() yet.

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

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

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

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

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

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

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

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.

```
class  NiceThread  extends Thread {
   public void run() {
      try {
         while ( !isInterrupted() ) {
            System.out.println(  "From: " + getName()  );
            sleep( 5000 );
         }
         System.out.println( "Clean up operations" );
      } catch ( InterruptedException interrupted ) {
         System.out.println( "In catch" );
      }
   }
}

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

Output

```
From: Thread-1
In catch
```

Who Sends sleep() is Important

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

```
public class Test  {
   public static void main( String args[] ) {
      try {
         NiceThread  missManners  = new NiceThread( );
         missManners.setPriority( 1 );
         missManners.start();
         missManners.sleep( 5000);
         missManners.interrupt();
      } catch ( InterruptedException interrupted ) {
         System.out.println( "Caught napping" );
      }
   }
}

class  NiceThread  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" );
      }
   }
}
```

Output

```
From: Thread-1
From: Thread-1
From: Thread-1
From: Thread-1
Clean up operations
```

IO Blocks

A read() on an inputstream or reader blocks. Once a thread calls read() it will not respond to interrupt() (or much else) until the read is completed. This is a problem when a read could take a long time: reading from a socket or the keyboard. If the input is not forth coming, the read() could block forever.

Nonblocking IO on Sockets

Set the SoTimeout on the socket before reading

Then the read will time out and exit with java.io.InterruptedIOException

InputStream is still usable

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.

```
class  AccessExample {
   int[]  data;
   int safeInt;

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

   public  void safeCode( int size,  int  startValue){
      int[]  verySafe  =  new  int[ size ];

      for  ( int  index  = 0; index < size;  index++ )
         verySafe[ index ]  =  (int ) Math.sin( index * startValue );
   }

   public void setInt( int aValue ) {
      safeInt = aValue;
   }

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

Mutual Access Problem

```
class Trouble extends Thread {
   int size;
   int startValue;
   AccessExample data;

   public Trouble( int aSize, int aStartValue, AccessExample myData ) {
      size = aSize;
      startValue = aStartValue;
      data = myData;
   }

   public void run()   {
      for (int k = 0; k < 10; k++ ) {
         data.setInt( size);
         data.safeCode( size, startValue );
         data.dangerousCode( size, startValue);
      }
   }
}
public class Test  {
   public static void main( String args[] ) throws Exception {
   AccessExample someData = new AccessExample();
   Trouble one = new Trouble( 500000, 0, someData );
   Trouble two = new Trouble( 3, 22, someData );
   one.start();
   two.start();
   two.join();
   one.join();
   System.out.println( someData );
   }
}
```

Output

```
rohan 31-> j2 -native Test
java.lang.ArrayIndexOutOfBoundsException: 3
        at AccessExample.dangerousCode(Compiled Code)
        at Trouble.run(Compiled Code)
array length 3 array values 0
```

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 Test  {
   public static void main( String args[] ) throws Exception {
      LockTest aLock = new LockTest();
      TryLock tester = new TryLock( aLock );
      tester.start();

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

class TryLock extends Thread {
   private LockTest myLock;

   public TryLock( LockTest aLock ) {
      myLock = aLock;
   }
   public void run()   {
      System.out.println( "Start run");
      myLock.enter();
      System.out.println( "End run");
   }
}

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

Output

```
In Block
Start run
End Block
In enter
End run
```

Deadlock

The following code creates a deadlock

```
class  Friendly  extends Thread {
   private  Friendly  aFriend;

   public  Friendly(  String  name ) {      super( name ); }

   public void setFriend( Friendly myFriend )
      { aFriend = myFriend;}

   public synchronized void hug() {
      try {
         System.out.println(  "I " + getName() + " am hugged ");
         sleep( 5 );
         aFriend.rehug();
      } catch ( InterruptedException notInThisExample ){}
   }

   public synchronized void rehug() {
      System.out.println( "I " + getName() + " am rehugged ");
   }

   public  void  run()   {aFriend.hug(); }
}

public class Test {
   public static void main( String args[] )  {
      Friendly  fred  =  new Friendly("Fred");
      Friendly  sam = new Friendly( "Sam");
      fred.setFriend( sam );
      sam.setFriend( fred );
      fred.start();
      sam.start();
      System.out.println( "End" );
   }
}
```

Output

```
End
I Fred am hugged
I Sam am hugged
```

Deadlock Avoided

Here we show how to avoid the deadlock of the previous slide.

```
class  Friendly  extends Thread      {
   private  Friendly  aFriend;
   private Object lock;

   public  Friendly(  String  name, Object lock )     {
      super( name );
      this.lock = lock;
   }

   public void setFriend( Friendly myFriend ) {
      aFriend = myFriend;
   }

   public synchronized void hug()   {
      try {
         System.out.println(  "I " + getName() + " am hugged ");
         sleep( 5 );
         aFriend.rehug();
      }
      catch ( InterruptedException notInThisExample ) {}
   }

   public synchronized void rehug(){
      System.out.println( "I " + getName() + " am rehugged ");
   }

   public  void  run() {
      synchronized ( lock) {
         aFriend.hug();
      }
   }
}
```

//Deadlock Avoided Continued

```
public class Test
  {

  public static void main( String args[] ) //throws Exception
    {
    Object aLock = "Schalage";
    Friendly  fred  =  new Friendly("Fred", aLock);
    Friendly  sam = new Friendly( "Sam", aLock);
    fred.setFriend( sam );
    sam.setFriend( fred );
    fred.start();
    sam.start();
    System.out.println( "End" );
    }
  }
```

Output

```
End
I Sam am hugged
I Fred am rehugged
I Fred am hugged
I Sam am rehugged
```

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
      }
```

Volatile

Java allows threads that access shared variables to keep private working copies of the variables. This improves the performance of multiple threaded programs. These working copies are reconciled with the master copies in shared main memory when objects are locked or unlocked. If you do not wish to use synchronized, Java has a second method to make sure that threads are using the proper value of shared variables. If a field is declared volatile, then a thread must reconcile its working copy of the field every time it accesses the variable. Operations on the master copy of the variable are performed in exactly the order that the thread requested. In the example on the next slide, a threads copy of the field "value" can get out of synch with its actual value.

Volatile Example

```
class ExampleFromTheBook {
   int value;
   volatile int volatileValue;

   public void setValue( int newValue ) {
      value = newValue;
      volatileValue = newValue;
   }

   public void display() {
      value = 5;
      volatileValue = 5;

      for ( int  k = 0; k < 5; k++ ) {
         System.out.println( "Value " + value );
         System.out.println("Volatile " + volatileValue );
         Thread.yield( );
      }
   }
}
class ChangeValue extends Thread {
   ExampleFromTheBook myData;

   public ChangeValue( ExampleFromTheBook data ) {
      myData = data;
   }

   public void run() {
      for ( int  k = 0; k < 5; k++ ) {
         myData.value = k;
         myData.volatileValue = k;
         Thread.yield( );
      }
   }
}
```

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

   ExampleFromTheBook example = new ExampleFromTheBook();
   ChangeValue changer = new ChangeValue( example );
   changer.start();

   example.display();
   }
}
```

Some of the Output

```
Value 5
Volatile 1
Value 2
Volatile 2
Value 3
Volatile 3
```

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.Vector;

class Queue {
   Vector elements = new Vector();
   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

```
class Producer extends Thread
   {
   Queue factory;
   int workSpeed;

   public Producer( String name, Queue 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

```
class Test
  {
  public  static  void  main( String  args[] ) throws Exception
    {
    Queue wallmart = new Queue();
    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

Thread Pools

Threads are expensive to start, so a server may keep a set of threads waiting for work to do. However, keeping a thread around does have some expense. Also, don't forget that once a thread's run method is done, the thread is dead.

```
import java.util.List;
public class SquareThreads extends Thread {
  List pool;

  public SquareThreads( List taskPool ) {
    pool = taskPool;
  }

  public void run() {
    Integer toSquare;
    while (true) {
      synchronized (pool) {
        while (pool.isEmpty() )
          try {
            pool.wait();
          }
          catch (java.lang.InterruptedException error) {
            return; // no clean up to do
          }
        toSquare = (Integer) pool.remove( pool.size() - 1);
      }
      System.out.println( "Thread " + getName() + " result is: " +
        (toSquare.intValue() * toSquare.intValue()));
    }
  }
}
```

Running the Example

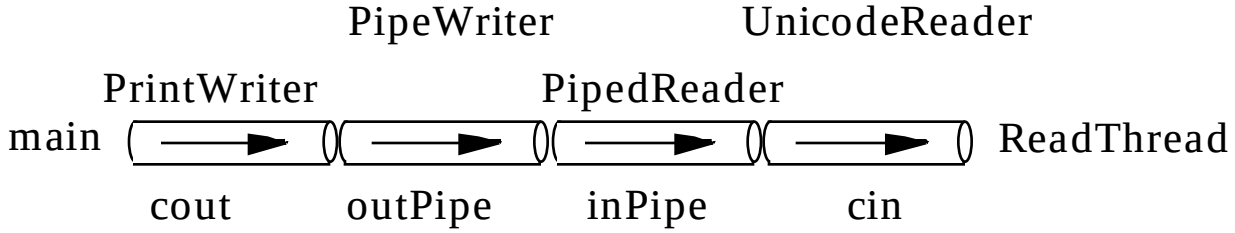
```
import java.util.Vector;

public class PoolExample {

   public static void main(String args[]) {
      Vector pool = new Vector();
      SquareThreads[] threads = new SquareThreads[3];
      for (int k = 0; k < threads.length;k++)
         {
         threads[k] = new SquareThreads( pool);
         threads[k].start();
         }
      for (int k = 0; k < 10; k++ )
         {
         synchronized (pool)
            {
            pool.add( new Integer( k));
            pool.notifyAll();
            }
         }
   }
}
```

Output

```
Thread Thread-1 result is: 64
Thread Thread-1 result is: 36
Thread Thread-2 result is: 49
Thread Thread-0 result is: 81
Thread Thread-1 result is: 25
Thread Thread-2 result is: 16
Thread Thread-0 result is: 9
Thread Thread-1 result is: 4
Thread Thread-2 result is: 1
Thread Thread-0 result is: 0
```

Piped Streams & Threads

In most streams, one end of the stream is "connected" to a file, socket, keyboard, etc. With piped streams, both ends are in your program. This allows one thread to write data via a stream to another thread in your program. The following example illustrates this.




```
class  TestIO {
  public static void main( String[]  args )  throws IOException {

     PipedReader  inPipe  =  new PipedReader();
     PipedWriter  outPipe  =  new PipedWriter(inPipe);

     PrintWriter  cout  =  new  PrintWriter(  outPipe  );

     ReadThread  reader  =  new ReadThread( "Read",  inPipe  );
     reader.setPriority( 6 );   // 5 is normal priority
     reader.start();

     System.out.println( "In Main" );
     cout.println( "Hello" );
     System.out.println( "End" );
  }
}
```

Messages between Threads

```
import java.io.*;
import sdsu.io.UnicodeReader;

class  ReadThread  extends Thread {
  private  UnicodeReader  cin;

  public  ReadThread(  String  name,  PipedReader  input ) {
    super( name );
    cin  =  new UnicodeReader( input );
  }

  public  void run() {
    try {
      System.out.println( "Start " + getName() );
      String  message  =  cin.readWord();
      System.out.println( message +  " From: " + getName() );
    } catch  ( Exception  ignored ) {}
  }
}
```

Output

```
Start Read
In Main
End
Hello From: Read
```

Debugging Threads

Some useful methods in Thread for debugging

public static void **dumpStack**()

Prints a stack trace for the current thread on System.out

public String **toString**()

Returns a String representation of the Thread, including the thread's name, priority and thread group.

public int **countStackFrames**()

Returns the number of stack frames in this Thread. The Thread must be suspended when this method is called.

Some Thread Ideas
Passing Data

When we pass data in or out of a method, there are problems with the data being changed by another thread while the method is using the data.

```
public int[] arrayPartialSums( int[] input ) {
   for ( int k = 1; k < input.length; k ++ )
      input[k] = input[ k - 1] + input[ k ];
   return input;
}
```

In the method below, even if all the methods of Foo are synchronized another thread can change the state of aFoo while objectMethod is executing.

```
public Object objectMethod( Foo aFoo ) {
   aFoo.bar();
   aFoo.process();
   return aFoo().getResult();
}
```

Lock the Data

If all users of aFoo follow the convention of locking the object before using it, then a Foo will not change in objectMethod due to activities in other threads.

```
public Object objectMethod( Foo aFoo ) {
   synchronized ( aFoo ) {
      aFoo.bar();
      aFoo.process();
      return aFoo().getResult();
   }
}
```

Clone the Data in the Method

Creating a clone helps insure that the local copy will not be modified by other threads. Of course, you need to perform a deep copy to insure no state is modified by other threads.

```
public int[] arrayPartialSums( int[] input ) {
   int[] inputClone;
   synchronized (input) {
      inputClone = input.clone();
   }

   for ( int k = 1; k < input.length; k ++ )
      inputClone [k] = inputClone [ k - 1] + inputClone [ k ];
   return inputClone;
}
```

Pass in a Clone

```
public void callerMethod() {
   // blah

   aWidget.arrayPartialSums( intArray.clone() )
}
```

Passer nulls its Reference

If the calling method removes its copy of parameters, then there should only be one copy of the parameter.

```
public void callerMethod() {
   // blah

   aWidget.arrayPartialSums( intArray )
   intArray = null;
}
```

Returner nulls its Reference

If a method nulls out its copy of values it returns or returns a clone, it will reduce the problem of two threads accessing the same reference.

```
public Foo aMethod() {
  Foo localVarCopy = theRealFooReference;
  theRealFooReference = null;
  return localVarCopy;
}

public Foo aMethod() {
  return theRealFooReference.clone();
}
```

Immutable Objects

Designing classes so the state of the object can not be modified eliminates the problem of multiple threads modifying objects state. String is an example of this.

A weaker idea is to create read-only copies of existing objects. An even weaker idea is to create read-only wrappers for existing objects. The later can be strengthened by using in with the clone method. The following two slides illustrate read-only objects.

Read-Only Copies - Inheritance Version

```
public class Point {
   int x;
   int y;

   public Point( int x , int y ) {
      this.x = x;
      this.y = y;
   }

   public int y() { return y; }

   public void y( int newY) { y = newY; }

   public int x() { return x; }

   public void x( int newX) { x = newX; }
}

public class ReadOnlyPoint extends Point {
   public ReadOnlyPoint( int x, int y ) {
      super( x, y );
   }

   public ReadOnlyPoint( Point aPoint ) {
      super( aPoint.x(), aPoint.y() );
   }

   public void y( int newY ) {
      throw new UnsupportedOperationException() ;
   }

   public void x( int newX ) {
      throw new UnsupportedOperationException() ;
   }
}
```

Read-Only Wrappers - Composition Version

```
interface Point {
   public int y();
   public void y( int newY);
   public int x();
   public void x( int newX);
}

public class ReadWritePoint implements Point {
   int x;
   int y;

   public ReadWritePoint( int x , int y ) {
      this.x = x;
      this.y = y;
   }

   public int y() { return y; }
   public void y( int newY) { y = newY; }
   public int x() { return x; }
   public void x( int newX) { x = newX; }
}

public class ReadOnlyWrapperPoint implements Point {
   Point myData;

   public ReadOnlyWrapperPoint( Point aPoint ) {myData = aPoint; }

   public int y() { return myData.y(); }
   public int x() { return myData.x(); }
   public void y( int newY ) {
      throw new UnsupportedOperationException() ;
   }
   public void x( int newX ) {
      throw new UnsupportedOperationException() ;
   }
}
```

Multiple Versions of Data Structures

We may needs different versions of a data structure that works differently if it is uses sequentially or with threads. On this slide, we have a Stack that is not synchronized for use in sequential programming. Composition is used over inheritance. Since we may need a LinkedListStack class, composition will allow the SynchronizedStack and the WaitingStack to work with LinkedListStack objects.

```
interface Stack  {
   public void push( float item );
   public float pop();
   public boolean isEmpty();
   public boolean isFull();
}

public class ArrayStack implements Stack {

   private float[] elements;
   private int topOfStack = -1;

   public ArrayStack( int stackSize )  {
      elements = new float[ stackSize ];
   }

   public void push( float item )  {
      elements[ ++topOfStack ] = item;
   }

   public float pop()  {
      return elements[ topOfStack-- ];
   }

   public boolean isEmpty()  {
      if ( topOfStack < 0 ) return true;
      else            return false;
   }

   public boolean isFull()  {
      if ( topOfStack >= elements.length )       return true;
      else                              return false;
   }
}
```

The Synchronized Stack

This example provides straightforward synchronization for a Stack object.

```
public class SynchonizedStack implements Stack {
   Stack myStack;

   public SynchonizedStack() {
      this( new ArrayStack() );
   }

   public SynchonizedStack( Stack aStack )  {
      myStack = aStack;
   }

   public synchonized boolean isEmpty() {
      return myStack.IsEmpty();
   }

   public synchonized boolean isFull() {
      return myStack.isFull();
   }

   public synchonized void push( float item )  {
      myStack.push( item );
   }

   public synchonized float pop()  {
      return myStack.pop();
   }
}
```

WaitingStack

In sequential programming there is not much that can be done when you attempt to pop() an element off an empty stack. In concurrent programming, we can have the thread that requested the pop() wait until another thread pushes an element on the stack. The stack below does this.

```
public class WaitingStack implements Stack {
   Stack myStack;

   public WaitingStack( Stack aStack )  {
      myStack = aStack;
   }

   public synchonized boolean isEmpty() {
      return myStack.IsEmpty();
   }

   public synchonized boolean isFull() {
      return myStack.isFull();
   }

   public synchonized void push( float item )  {
      myStack.push( item );
      notifyAll();
   }

   public synchonized float pop()  {
      while ( isEmpty() )
         try {
            wait();
         } catch ( InterruptedException ignore ) {}
      return myStack.pop();
   }
}
```

Background Operations

There are times when we would like to perform operations in the "background". When these operations are done then another thread will use the result of the computations. How do we know when the background thread is done? The polling done here does consume CPU cycles. We could end up with one thread wasting CPU time just checking if another thread is done.

```
class TimeConsumingOperation extends Thread {
  Object result;
  boolean isDone = false;

  public void run() {
    DownLoadSomeData&PerformSomeComplexStuff;
    result = resultOfMyWork;
    isDone = true;
  }

  public Object getResult() {
    return result;
  }

  public boolean isDone() {
    return isDone();
  }
}

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

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

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

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

Futures

One way to handle these "background" operations is to wrap them in a sequential appearing class: a future. When you create the future object, it starts the computation in a thread. When you need the result, you ask for it. If it is not ready yet, you wait until it is ready.

```
class FutureWrapper {
   TimeConsumingOperation myOperation;

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

   public Object compute() {
      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.compute();
   }
}
```

Callbacks

The thread doing the computation can use callbacks to notify other objects that it is done.

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

  public void exceptionCallback( Exception problem ) {
    handleException;
  }

  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() {
    try {
      DownLoadSomeData;
      PerformSomeComplexStuff;
      master.normalCallback( resultOfMyWork );
    } catch ( Exception someProblem ) {
      master.exceptionCallback( someProblem );
    }
  }
}
```

Callbacks with Listeners

The following code uses Java's standard idea of listeners to generalize the callback process. Anyone that is interested in the results of the thread implements the ThreadListener interface and registers their interest (shown later). The results are passed back in a ThreadEvent object.

```
public interface ThreadListener {
   public void threadResult( ThreadEvent anEvent );
   public void threadExceptionThrown( ThreadEvent anEvent );
}

public class ThreadEvent extends java.util.EventObject {
   Exception thrown;
   Object result;

   public ThreadEvent( Object source ) {
      super( source );
   }

   public ThreadEvent( Object source, Object threadResult ) {
      super( source );
      result = threadResult;
   }

   public ThreadEvent( Object source, Exception threadException ) {
      super( source );
      thrown = threadException;
   }

   public Exception getException() {
      return thrown;
   }

   public Object getResult() {
      return result;
   }
}
```

ThreadListenerHandler

ThreadListenerHandler is a helper class used to perform the actual broadcast.

```
public class ThreadListenerHandler {
   ArrayList listeners = new ArrayList();
   Object theListened;

   public ThreadListenerHandler( Object listened ) {
      theListened = listened;
   }

   public synchronized void addThreadListener( ThreadListener aListener ) {
      listeners.add( aListener );
   }

   public synchronized void removeThreadListener( ThreadListener aListener ) {
      listeners.remove( aListener );
   }

   public void broadcastResult( Object result ) {
      Iterator sendList;
      synchronized ( this ) {
         sendList = ( (ArrayList ) listeners.clone()).iterator();
      }

      ThreadEvent broadcastData = new ThreadEvent( theListened, result );

      while ( sendList.hasNext() ) {
         ThreadListener aListener = (ThreadListener) sendList.next();
         aListener.threadResult( broadcastData );
      }
   }

   public void broadcastException( Exception anException ) {
      Iterator sendList;
      synchronized ( this ) {
         sendList = ( (ArrayList ) listeners.clone()).iterator();
      }
      ThreadEvent broadcastData = new ThreadEvent( theListened, anException);

      while ( sendList.hasNext() ) {
         ThreadListener aListener = (ThreadListener) sendList.next();
         aListener.threadExceptionThrown( broadcastData );
      }
   }
}
```

TimeConsumingOperation

The methods addThreadListener and removeThreadListener are used by client code to register interest in "listening" to this thread.

```
class TimeConsumingOperation extends Thread {

   ThreadListenerHandler listeners =
      new ThreadListenerHandler( this );

   public void addThreadListener( ThreadListener aListener ) {
      listeners.addThreadListener( aListener );
   }

   public void removeThreadListener( ThreadListener aListener ) {
      listeners.removeThreadListener( aListener );
   }

   public void run() {
      try {
         DownLoadSomeData;
         PerformSomeComplexStuff;
         listeners.broadcastResult( null );

      } catch ( Exception someProblem ) {
         listeners.broadcastException( someProblem );
      }
   }
}
```

MasterThread

Here we can see how the creator of TimeConsumingOperation works.

```
class MasterThread implements ThreadListener {

   public void threadResult( ThreadEvent threadResult ) {
      // Get the results and use them to do perform the task
      threadResult.getResult();
   }

   public void threadExceptionThrown( ThreadEvent problem ) {
      // The other thread ended in an exception, deal with that here
      problem.getException();
   }

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

      // Register interest in the background's results
      backGround.addThreadListener( this );

      backGround.start();
      moreComputation;
   }
}
```

Using an Adapter

Sometimes you may not want your class to implement the ThreadListener interface. Other method names and parameter types may be more appropriate for your context. We can use an "adapter" class to adapt the methods in the MasterThread class to the methods in the ThreadListener interface. This use of anonymous classes is a major motivation for adding anonymous classes to Java.

```
class MasterThread {

  public void compute( String data ) {
    UseStringToPerformComputation
  }

  public void handleException( Exception problem ) {
    HandleTheException
  }

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

    backGround.addThreadListener( new ThreadListener() {
        public void threadResult( ThreadEvent anEvent ) {
          compute( (String) anEvent.getResult() );
        }
        public void threadExceptionThrown(ThreadEvent anEvent ) {
          handleException( anEvent.getException() );
        }
      }
    );

    backGround.start();
    moreComputation;
  }
}
```