core WEB programming

Multithreaded Programming

Agenda

- **Why threads?**
- **Approaches for starting threads**
  - Separate class approach
  - Callback approach
- **Solving common thread problems**
- **Synchronizing access to shared resources**
- **Thread life cycle**
- **Stopping threads**

Concurrent Programming Using Java Threads

- **Motivation**
  - Efficiency
    - Downloading network data files
  - Convenience
    - A clock icon
  - Multi-client applications
    - HTTP Server, SMTP Server
- **Caution**
  - Significantly harder to debug and maintain
- **Two Main Approaches:**
  - Make a self-contained subclass of `Thread` with the behavior you want
  - Implement the `Runnable` interface and put behavior in the run method of that object

Thread Mechanism One: Making a Thread Subclass

- **Create a separate subclass of Thread**
  - No import statements needed: Thread is in java.lang
- **Put the actions to be performed in the run method of the subclass**
  - public void run() { … }
- **Create an instance of your Thread subclass**
  - Or lots of instances if you want lots of threads
- **Call that instance's start method**
  - You put the code in run, but you call start!

Thread Mechanism One: Making a Thread Subclass

```
public class DriverClass extends SomeClass {
  ...
  public void startAThread() {
    // Create a Thread object
    ThreadClass thread = new ThreadClass();
    // Start it in a separate process
    thread.start();
  }
}
```

```
public class ThreadClass extends Thread {
  public void run() {
    // Thread behavior here
  }
}
```


Thread Mechanism One: Example

```
public class Counter extends Thread {
  private static int totalNum = 0;
  private int currentNum, loopLimit;

  public Counter(int loopLimit) {
    this.loopLimit = loopLimit;
    currentNum = totalNum++;
  }

  private void pause(double seconds) {
    try { Thread.sleep(Math.round(1000.0*seconds)); }
    catch(InterruptedException ie) {}
  }

  ...
```

Thread Mechanism One: Example (Continued)

```
  ...

  /** When run finishes, the thread exits. */

  public void run() {
    for(int i=0; i<loopLimit; i++) {
      System.out.println("Counter " + currentNum
                         + ": " + i);
      pause(Math.random()); // Sleep for up to 1 second
    }
  }
}
```


Thread Mechanism One: Example (Continued)

```
public class CounterTest {
  public static void main(String[] args) {
    Counter c1 = new Counter(5);
    Counter c2 = new Counter(5);
    Counter c3 = new Counter(5);
    c1.start();
    c2.start();
    c3.start();
  }
}
```

Thread Mechanism One: Result

```
Counter 0: 0
Counter 1: 0
Counter 2: 0
Counter 1: 1
Counter 2: 1
Counter 1: 2
Counter 0: 1
Counter 0: 2
Counter 1: 3
Counter 2: 2
Counter 0: 3
Counter 1: 4
Counter 0: 4
Counter 2: 3
Counter 2: 4
```


Thread Mechanism Two: Implementing Runnable

- **Put the actions to be performed in the run method of your existing class**
- **Have class implement Runnable interface**
  - If your class already extends some other class (e.g., Applet), why can't it still extend Thread? Because Java does not support multiple inheritance.
- **Construct an instance of Thread passing in the existing object (i.e., the Runnable)**
  - Thread t = new Thread(theRunnableObject);
- **Call that Thread's start method**
  - t.start();

Thread Mechanism Two: Implementing Runnable (Cont.)

```
public class ThreadedClass extends AnyClass
                           implements Runnable {
  public void run() {
    // Thread behavior here
    // If you want to access thread instance
    // (e.g. to get private per-thread data), use
    // Thread.currentThread().
  }

  public void startThread() {
    Thread t = new Thread(this);
    t.start(); // Calls back to run method in this
  }
  ...
}
```


Thread Mechanism Two: Example

```
public class Counter2 implements Runnable {
  private static int totalNum = 0;
  private int currentNum, loopLimit;

  public Counter2(int loopLimit) {
    this.loopLimit = loopLimit;
    currentNum = totalNum++;
    Thread t = new Thread(this);
    t.start();
  }
  ...
```

Thread Mechanism Two: Example (Continued)

```
  ...
  private void pause(double seconds) {
    try { Thread.sleep(Math.round(1000.0*seconds)); }
    catch(InterruptedException ie) {}
  }

  public void run() {
    for(int i=0; i<loopLimit; i++) {
      System.out.println("Counter " + currentNum
                         + ": " + i);
      pause(Math.random()); // Sleep for up to 1 second
    }
  }
}
```


Thread Mechanism Two: Example (Continued)

```
public class Counter2Test {
  public static void main(String[] args) {
    Counter2 c1 = new Counter2(5);
    Counter2 c2 = new Counter2(5);
    Counter2 c3 = new Counter2(5);
  }
}
```

Thread Mechanism Two: Result

```
Counter 0: 0
Counter 1: 0
Counter 2: 0
Counter 1: 1
Counter 1: 2
Counter 0: 1
Counter 1: 3
Counter 2: 1
Counter 0: 2
Counter 0: 3
Counter 1: 4
Counter 2: 2
Counter 2: 3
Counter 0: 4
Counter 2: 4
```


Race Conditions: Example

```
public class BuggyCounterApplet extends Applet
                                implements Runnable{
  private int totalNum = 0;
  private int loopLimit = 5;

  public void start() {
    Thread t;
    for(int i=0; i<3; i++) {
      t = new Thread(this);
      t.start();
    }
  }

  private void pause(double seconds) {
    try { Thread.sleep(Math.round(1000.0*seconds)); }
    catch(InterruptedException ie) {}
  }
  ...
```

Race Conditions: Example (Continued)

```
  ...
  public void run() {
    int currentNum = totalNum;
    System.out.println("Setting currentNum to "
                       + currentNum);
    totalNum = totalNum + 1;
    for(int i=0; i<loopLimit; i++) {
      System.out.println("Counter "
                         + currentNum + ": " + i);
      pause(Math.random());
    }
  }
}
```

- **What's wrong with this code?**

Race Conditions: Result

- **Usual Output**

```
Setting currentNum to 0
Counter 0: 0
Setting currentNum to 1
Counter 1: 0
Setting currentNum to 2
Counter 2: 0
Counter 2: 1
Counter 1: 1
Counter 0: 1
Counter 2: 2
Counter 0: 2
Counter 1: 2
Counter 1: 3
Counter 0: 3
Counter 2: 3
Counter 1: 4
Counter 2: 4
Counter 0: 4
```

- **Occasional Output**

```
Setting currentNum to 0
Counter 0: 0
Setting currentNum to 1
Setting currentNum to 1
Counter 0: 1
Counter 1: 0
Counter 1: 0
Counter 0: 2
Counter 0: 3
Counter 1: 1
Counter 0: 4
Counter 1: 1
Counter 1: 2
Counter 1: 3
Counter 1: 2
Counter 1: 3
Counter 1: 4
Counter 1: 4
```

Race Conditions: Solution?

- **Do things in a single step**

```
public void run() {
  int currentNum = totalNum++;
  System.out.println("Setting currentNum to "
                     + currentNum);
  for(int i=0; i<loopLimit; i++) {
    System.out.println("Counter "
                       + currentNum + ": " + i);
    pause(Math.random());
  }
}
```


Arbitrating Contention for Shared Resources

- **Synchronizing a Section of Code**

```
synchronized(someObject) {
  code
}
```

- **Normal interpretation**
  - Once a thread enters the code, no other thread can enter until the first thread exits.
- **Stronger interpretation**
  - Once a thread enters the code, no other thread can enter any section of code that is synchronized using the same "lock" tag

Arbitrating Contention for Shared Resources

- **Synchronizing an Entire Method**

```
public synchronized void someMethod() {
  body
}
```

- **Note that this is equivalent to**

```
public void someMethod() {
  synchronized(this) {
    body
  }
}
```


Common Synchronization Bug

- **What's wrong with this class?**

```
public class SomeThreadedClass extends Thread {
  private static RandomClass someSharedObject;
  ...
  public synchronized void doSomeOperation() {
    accessSomeSharedObject();
  }
  ...
  public void run() {
    while(someCondition) {
      doSomeOperation();      // Accesses shared data
      doSomeOtherOperation();// No shared data
    }
  }
}
```

Synchronization Solution

- **Solution 1: synchronize on the shared data**

```
public void doSomeOperation() {
  synchronized(someSharedObject) {
    accessSomeSharedObject();
  }
}
```

- **Solution 2: synchronize on the class object**

```
public void doSomeOperation() {
  synchronized(SomeThreadedClass.class) {
    accessSomeSharedObject();
  }
}
```

  - Note that if you synchronize a static method, the lock is the corresponding Class object, not `this`

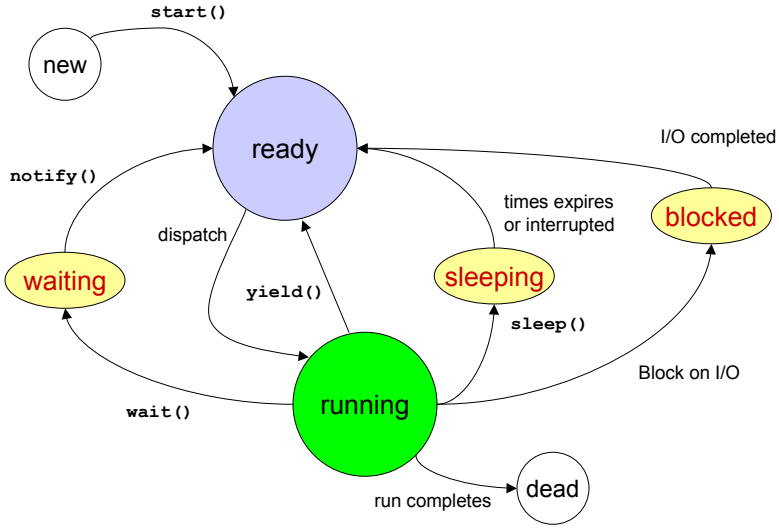
Synchronization Solution (Continued)

- **Solution 3: synchronize on arbitrary object**

```
public class SomeThreadedClass extends Thread {
  private static Object lockObject
    = new Object();
  ...
  public void doSomeOperation() {
    synchronized(lockObject) {
      accessSomeSharedObject();
    }
  }
  ...
}
```

  - Why doesn't this problem usually occur with Runnable?

Thread Lifecycle



Useful Thread Constructors

- **Thread()**
  - Default version you get when you call constructor of your custom Thread subclass.
- **Thread(Runnable target)**
  - Creates a thread, that, once started, will execute the `run` method of the target
- **Thread(ThreadGroup group, Runnable target)**
  - Creates a thread and places it in the specified thread group
  - A `ThreadGroup` is a collection of threads that can be operated on as a set
- **Thread(String name)**
  - Creates a thread with the given name
  - Useful for debugging

Thread Priorities

- **A thread's default priority is the same as the creating thread**
- **Thread API defines three thread priorities**
  - `Thread.MAX_PRIORITY` (typically 10)
  - `Thread.NORM_PRIORITY` (typically 5)
  - `Thread.MIN_PRIORITY` (typically 1)
- **Problems**
  - A Java thread priority may map differently to the thread priorities of the underlying OS
    - Solaris has $2^{32}-1$ priority level; Windows NT has 7 user priority levels
  - Starvation can occur for lower-priority threads if the higher-priority threads never terminate, sleep, or wait for I/O

Useful Thread Methods

- **currentThread**
  - Returns a reference to the currently executing thread
  - This is a `static` method that can be called by arbitrary methods, not just from within a `Thread` object
    - I.e., anyone can call Thread.currentThread
- **interrupt**
  - One of two outcomes:
    - If the thread is executing `join`, `sleep`, `or wait`, an `InterruptedException` is thrown
    - Sets a flag, from which the interrupted thread can check (isInterrupted)
- **interrupted**
  - Checks whether the currently executing thread has a request for interruption (checks flag) and clears the flag

Useful Thread Methods (Continued)

- **isInterrupted**
  - Simply checks whether the thread's interrupt flag has been set (does not modify the flag)
    - Reset the flag by calling `interrupted` from within the `run` method of the flagged thread
- **join**
  - Joins to another thread by simply waiting (sleeps) until the other thread has completed execution
- **isDaemon/setDaemon**
  - Determines or set the thread to be a daemon
  - A Java program will exit when the only active threads remaining are daemon threads

Useful Thread Methods (Continued)

- **start**
  - Initializes the thread and then calls `run`
  - If the thread was constructed by providing a `Runnable`, then start calls the run method of that `Runnable`
- **run**
  - The method in which a created thread will execute
  - Do not call run directly; call start on the thread object
  - When run completes the thread enters a dead state and cannot be restarted

Useful Thread Methods (Continued)

- **sleep**
  - Causes the currently executing thread to do a nonbusy wait for at least the amount of time (milliseconds), unless interrupted
  - As a static method, may be called for nonthreaded applications as well
    - I.e., anyone can call Thread.sleep
    - Note that sleep throws InterruptedException. Need try/catch
- **yield**
  - Allows any other threads of the same or higher priority to execute (moves itself to the end of the priority queue)
  - If all waiting threads have a lower priority, then the yielding thread remains on the CPU

Useful Thread Methods (Continued)

- **wait/waitForAll**
  - Releases the lock for other threads and suspends itself (placed in a wait queue associated with the lock)
  - Thread can be restarted through `notify` or `notifyAll`
  - These methods must be synchronized on the lock object of importance
- **notify/notifyAll**
  - Wakes up all threads waiting for the lock
  - A notified doesn't begin immediate execution, but is placed in the runnable thread queue

Stopping a Thread

```
public class ThreadExample implements Runnable {
  private boolean running;
  public ThreadExample()
     Thread thread = new Thread(this);
     thread.start();
  }
  public void run(){
    running = true;
    while (running) {
      ...
    }
    doCleanup();
  }

  public void setRunning(boolean running) {
    this.running = running;
  }
}
```


Signaling with wait and notify

```
public class ConnectionPool implements Runnable {
  ...
  public synchronized Connection getConnection() {
    if (availableConnections.isEmpty()) {
      try {
        wait();
      } catch(InterruptedException ie) {}
      // Someone freed up a connection, so try again.
      return(getConnection());
    } else {
      // Get available connection
      ...
      return(connection)
    }
  }
```

Signaling with wait and notify (Continued)

```
  public synchronized void free(Connection connection) {
    busyConnections.removeElement(connection);
    availableConnections.addElement(connection);
    // Wake up threads that are waiting
    // for a connection
    notifyAll();
  }
  ...
}
```


Summary

- **Achieve multithreaded behavior by**
  - Inheriting directly from `Thread` (separate class approach)
  - Implementing the `Runnable` interface (callback approach)
- **In either case, put your code in the `run` method. Call `start` on the Thread object.**
- **Avoid race conditions by placing the shared resource in a synchronized block**
- **You can't restart a dead thread**
- **Stop threads by setting a flag that the thread's run method checks**

core WEB programming

Questions?