Administrivia

- Exam handed back today
  - Mean and median: 65
  - 25%: 57  75%: 74
  - answers posted
- Java threads project posted
  - due Friday, Oct. 26
- Read Chapter 14 of Eckel on threads
  - don’t worry about GUI/Swing example details

Last time - Java

- Inner classes
  - nested classes/interfaces – behave like static
    method/instance variable
  - can only access statics of enclosing class
  - standard inner classes – behave like class
    method/variable
    - no statics allowed
  - method class – like a method variable
    - no statics allowed
  - anonymous class – defined in new expression

Multithreading and Synchronization

What is a thread?

- It’s a program counter and a stack
- All threads share the same memory space
  - take turns with the CPU, for a uni-processor
  - run concurrently, on a multiprocessor
- Example: Web browser
  - one thread for I/O
  - one thread for each file being downloaded
  - one thread to render web page
- Running thread might
  - yield, sleep, wait for I/O or notify, be pre-empted

Writing Multi-threaded Code

- Need to control which events can happen simultaneously
  - e.g., update and display methods for a class
- Usually only covered in OS/DB courses
  - so few programmers have lots of training
- Can get inconsistent results or deadlock
  - problems often not easily reproduced
- Easy to get multi-threading, without trying
  - Graphical User Interfaces (GUI’s)
  - Remote Method Invocation
Extending class Thread

- Can build a thread class by extending java.lang.Thread
- Must supply a public void run() method
- Start a thread by invoking the start() method
- When a thread starts, executes run()
- When run() returns, thread is finished/dead

Simple thread methods

- void start()
- boolean isAlive()
- void setDaemon(boolean on)
  - if only daemon threads running, VM terminates
- void setPriority(int newPriority)
  - thread scheduler might respect priority
- void join() throws InterruptedException
  - waits for a thread to die/finish

Simple static thread methods

- Apply to thread invoking the method
  - void yield()
  - void sleep(long milliseconds)
    throws InterruptedException
  - Thread currentThread()

Runnable interface

- Extending Thread means can’t extend any other class
- Instead implement Runnable
  - declares that the class has a void run() method
- Can construct a new Thread
  - and give it an object of type Runnable as an argument to the constructor
  - Thread(Runnable target)
  - Thread(Runnable target, String name)

Thread Example

```java
public class ThreadDemo implements Runnable {
    public void run() {
        for (int i = 5; i > 0; i--) {
            System.out.println(i);
            try { Thread.sleep(1000); }
            catch(InterruptedException e) { };
            System.out.println("exiting "+Thread.currentThread());
        }
    }

    public static void main(String[] args) {
        Thread t = new Thread(new ThreadDemo(),"Demo Thread");
        System.out.println("main thread: "+t);
        t.start();
        try { Thread.sleep(3000); } catch (InterruptedException e) { };
        System.out.println("exiting "+Thread.currentThread());
    }
}
```

InterruptedException

- A number of thread methods throw it
  - really means: wakeUpCall
- interrupt() sends a wakeUpCall to a thread
- Won’t disturb the thread if it is working
  - but if thread attempts to sleep
  - it will get immediately woken up
- Will also wake up the thread if it is already asleep
- Thrown by sleep(), join(), wait()
Be careful with threads

- Under some implementations of JVM
  - a thread stuck in a loop will never yield by itself
- Preemptive scheduling would guarantee it
  - but not supported on all platforms
- Put `yield()` into loops
- I/O has highest priority, so should be able to get time on CPU

Example - why synchronization?

class UnSyncTest extends Thread {
  String msg;
  public UnSyncTest(String s) {
    msg = s; start();
  }
  public void run() {
    System.out.println("[ " + msg);
    try { Thread.sleep(1000); }
    catch(InterruptedException e) {}
    System.out.println("] ");
  }
  public static void main(String [] args) {
    new UnSyncTest("Hello");
    new UnSyncTest("UnSynchronized");
    new UnSyncTest("World");
  }
}

Synchronization issues

- Locks
- `synchronized` statements and methods
- `wait` and `notify`
- Deadlock

Locks

- *All* objects can be locked
- Only one thread can hold a lock on an object
  - other threads block until they can acquire it
- If your thread already holds a lock on an object
  - can lock it a second time
  - object not unlocked until both locks released
- No way to only attempt to acquire a lock

Daemon threads

- A thread can be marked as a daemon thread
- By default, thread acquires status of thread that spawned it
- When no threads running except daemons
  - program execution terminates

Synchronized methods

- A method can be synchronized
  - add `synchronized` modifier before return type
- Obtains a lock on object referenced by `this`,
  before executing method
  - releases lock when method completes
- For a static `synchronized` method
  - locks the class object
Synchronized statement

- `synchronized (obj) { statements }`
- Obtains a lock on `obj` before executing statements in block
- Releases lock once block completes
- Provides finer grained control than synchronized method
- Allows locking arguments to a method

Synchronization example

class SyncTest extends Thread {
    String msg;
    public SyncTest(String s) {
        msg = s;
        start();
    }
    public void run() {
        synchronized (SyncTest.class) {
            System.out.print(msg);
            try { Thread.sleep(1000); } catch (InterruptedException e) { }
            System.out.println(msg);
        }
    }
    public static void main(String[] args) {
        new SyncTest("Hello");
        new SyncTest("Synchronized");
        new SyncTest("World");
    }
}

Wait and Notify

- Must be called inside `synchronized` method or block of statements
- `a.wait()`
  - gives up lock(s) on `a`
  - adds thread to wait set for `a`
  - suspends thread
- `a.wait(int m)`
  - limits suspension to `m` milliseconds

Wait and Notify (cont.)

- `a.notify()` resumes one thread from `a`'s wait list
  - and removes it from wait set
  - no control over which thread
- `a.notifyAll()` resumes all threads on `a`'s wait list
- resumed thread(s) must reacquire lock before continuing
- `wait` doesn’t give up locks on any other objects
  - e.g., acquired by methods that called this one
Producer/Consumer Example – Too Much Synchronization

```java
public class ProducerConsumer {
    private boolean ready = false;
    private Object obj;
    public ProducerConsumer() { }
    public ProducerConsumer(Object o) {
        obj = o;  ready = true;
    }
    synchronized void produce(Object o) {
        while (ready) wait();
        obj = o;  ready = true;
        notifyAll();
    }
    synchronized Object consume() {
        while (!ready) wait();
        ready = false;
        notify();
        return obj;
    }
}
```

Changed example – Attempt to refine synch.

```java
synchronized void produce(Object o) {
    while (ready) {
        synchronized (empty) {
            try {empty.wait();}
            catch (InterruptedException e) { }
        }
    }
    obj = o;  ready = true;
    synchronized (full) {
        full.notify();
    }
}
```

A Better Solution

```java
synchronized void produce(Object o) {
    while (ready) {
        synchronized (empty) {
            try {empty.wait();}
            catch (InterruptedException e) { }
        }
    }
    obj = o;  ready = true;
    synchronized (full) {
        full.notify();
    }
}
```

```
Use two objects, empty and full, to allow two different wait sets
```

notify() vs. notifyAll()

- Very tricky to use notify() correctly
  - notifyAll() generally much safer
- To use correctly, should have:
  - all waiters are equal
  - each notify only needs to wake up 1 thread
  - handle InterruptedException correctly

InterruptedException Example

- Threads t1 and t2 are waiting
- Thread t3 performs a notify
  - thread t1 is selected
- Before t1 can acquire lock, t1 is interrupted
- t1’s call to wait throws InterruptedException
  - t1 doesn’t process notification
  - t2 doesn’t wake up

Handling InterruptedException

```java
synchronized (this) {
    while (!ready) {
        try { wait();}
        catch (InterruptedException e) { }
        notify();
        throw e;  }
    // do whatever
}
```
Deadlock

• Quite possible to create code that deadlocks
  – Thread 1 holds lock on A
  – Thread 2 holds lock on B
  – Thread 1 is trying to acquire a lock on B
  – Thread 2 is trying to acquire a lock on A
  – Deadlock!
• Not easy to detect when deadlock has occurred
  – other than by the fact that nothing is happening

A common multi-threading bug

• Threads might cache values
• Obtaining a lock forces the thread to get fresh values
• Releasing a lock forces the thread to flush out all pending writes
• volatile variables are never cached
• sleep(...) doesn’t force fresh values
• Many compilers don’t perform these optimizations
  — but some do (Hotspot?)
• Problem might also occur with multiple CPUs

Guidelines to simple/safe multi-threaded programming

• Synchronize access to shared data
• Don’t hold a lock on more than one object at a time
  – could cause deadlock
• Hold a lock for as little time as possible
  – reduces blocking waiting for locks
• While holding a lock, don’t call a method you don’t understand
  – e.g., a method provided by someone else, especially if you can’t be sure what it locks

Guidelines (cont.)

• Have to go beyond these guidelines for more complex situations
  – but need to understand threading and synchronization well
• Recommended book for more details:
  – Concurrent Programming in Java, by Doug Lea

I/O Classes

• File
  – directories
    • if (isDirectory()) System.out.println(f.list());
  – interface FilenameFilter – allows selection of sublist
• OutputStream – byte stream going out
• Writer – character stream going out
• InputStream – byte stream coming in
• Reader – character stream coming in

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries

I/O and Utility Libraries
OutputStream - bytes

- base types
  - ByteArrayInputStream
  - FileOutputStream
  - PipedOutputStream – goes to PipedInputStream
  - SocketOutputStream (not public) – goes to TCP socket
- Filters – wrapped around an OutputStream
  - BufferedOutputStream
  - ObjectOutputStream (should implement FilterOutputStream) – serialization of object graph

InputStream - bytes

- base types
  - ByteArrayInputStream
  - FileInputStream
  - PipedInputStream
  - SocketInputStream (not public) – comes from TCP socket
- Filters – wrapped around InputStream
  - BufferedInputStream
  - PushedBackInputStream
  - ObjectInputStream
- SequenceInputStream - concatenate

Reader - characters

- base types
  - ByteArrayInputStream
  - FileOutputStream
  - PipedOutputStream
  - SocketInputStream (not public) – goes to TCP socket
- Filters – wrapped around an OutputStream
  - BufferedWriter
  - ObjectOutputStream (should implement FilterOutputStream)

Writer - characters

- base types
  - OutputStreamWriter
  - ObjectOutputStream
  - ObjectOutputStream (should implement FilterOutputStream) – serialization of object graph
- Filters
  - PrintWriter – supports print, println
  - BufferedWriter
- Convenience writers
  - wrap OutputStreamWriter around an OutputStream
  - FileWriter and PipedWriter

java.util

- Vector
- Dictionaries
- Enumerations and Bitsets
- Collection classes

Vector

- A list/vector abstraction
- Can insert/delete/modify elements anywhere in the list
- Can access by position
- Stack
  - extension of Vector
  - adds push, pop, peek and empty
Dictionaries

- **Dictionary**
  - an abstract class
  - represents key to value mapping
- **HashTable**
  - an implementation of Dictionary
  - grows as needed
  - can be saved to a file (serializable, implements deep toString)

Enumerations and Bitsets

- **Enumeration**
  - an interface
  - used in many places to return an enumeration
    - public boolean hasMoreElements()
    - public Object nextElement()
- **BitSet**
  - provides representation of a set as a bit vector
  - grows as needed (like HashTable)

Collection Classes

- **interface Collection**
  - **interface List**
    - class Vector (and Stack)
    - class ArrayList
    - class LinkedList – doubly linked
  - **interface Set**
    - class HashSet
    - interface SortedSet
      - class TreeSet
  - **interface Map** – dictionary-like structures
    - class HashMap – replaces HashTable
    - interface SortedMap
      - class TreeMap

Other libraries

- **java.lang.Math**
  - abstract final class – only static members
  - includes constants $e$ and $\pi$
  - includes static methods for trig, exponentiation, min, max, …
- **java.text**
  - text formatting tools
    - class MessageFormat provides printf/scanf functionality
  - lots of facilities for internationalization