CMSC 433, Spring 2002
Programming Language Technology and Paradigms
Java Threads

Alan Sussman
February 21, 2002

Administrivia
• Project 2 due Wednesday, Feb. 27
  – questions?
• Project 1 commentary due March 6

Last time - Java
• Inner classes
  – static, standard, method, anonymous
• Libraries
  – I/O - OutputStreamWriter, InputStreamReader, Filters (Buffered)
  – Vectors, Maps, Collections (interfaces List and Set)
• Multithreading
  – thread is a program counter and a runtime stack
  – need for synchronization across threads to control which events can happen at the same time
  – extend class Thread to get a class that can run as a thread
  * extends the run() method
  * invoke via Thread.start()
  – Thread methods – start(), join(), yield(), sleep(), 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)

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

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

Example - why synchronization?

```java
class UnSyncTest extends Thread {
    String msg;
    public UnSyncTest(String s) {
        msg = s;  start();
    }
    public void run() {
        System.out.println("[");
        try { Thread.sleep(1000); }  
        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

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