Concurrent Programming, Threads

Sara Sprenkle

July 25, 2006

Announcements

- Assignment 5 due on Thursday
- Assignment 6 out on Thursday
- Project 2 out … soon

- Quiz

Review: UI Design

Review: UI Design

- Controlling environment
  - Settings, options
- Consistency
- Design for Extremes
- Activity-based Planning
- Usability Testing
- Color

Review: Regular Expressions

- Character Classes
  - . : any character except line characters
  - \d : any digit
  - \D: any non-digit
  - [ ]: programmer-defined character classes
- Capturing group
  - Designated by ()

Review: Regular Expressions

- Some other special characters
  - ^: not ([^Z] ≠ [A-Y]), or the beginning of the line
  - $: end of the line
  - *: ≥0 occurrences of pattern
  - +: ≥1 occurrences of the pattern
  - ?: matches 0 or 1 occurrences of the pattern
- What if want to specify all but the ^ character?

Review: JUnit

- Scope of testing?
- What are benefits of JUnit?
- Where do tests go?

Project 1 Discussion

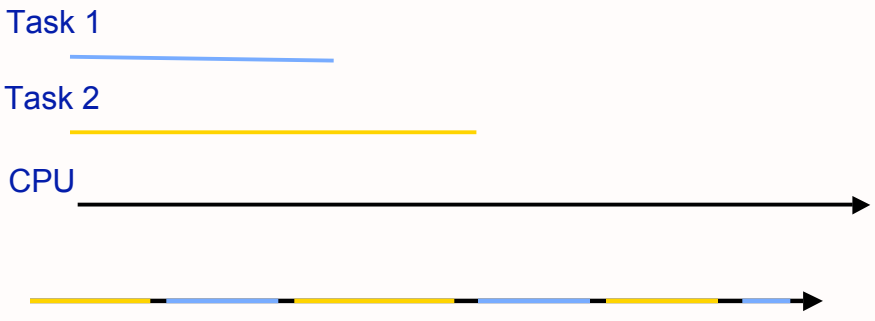
- Why was PileModel an inner class?
  - Did it have to be an inner class?
- How much of the original codebase did you need to understand?
  - At what level was your understanding for each class?
- How much of the original codebase did you modify?
- How did you design your code?

Multitasking

- Very common in modern operating systems
- Ability to have more than one program running at the same time
  - Or making it *look* like this is happening
- In reality, the operating system is distributing resources (most noticeably CPU time) to create this impression of parallel activity.
- Only one program is running at once
  - the OS simply switches from one to another in very quick switches

Multitasking

- What are goals of multitasking?
- When are "smart" times to switch between tasks?



Multitasking

- Goal: keep CPU completely utilized
- Policies
  - Assign priorities to threads
  - Switch when one task has to pause for I/O
  - Set time interval

Multithreading

- Process: a "program" that can be multitasked
  - To view the processes on your machine: ps or Task Manager
- Multithreaded programs (or concurrent programs)
  - one program can appear to be doing multiple things at one time
- Thread or thread of execution
  - A task; a light-weight process
- How is a thread different than a process?

Threads vs. Processes

- A process runs in its own context
  - Seems to have its own CPU, memory, registers, …
  - Data is not shared between processes
- A thread also seems to have its own CPU, execution stack, and registers
  - All of the threads in a program share the same variables and data
  - Threads run in the context of a program

Processes in UNIX

- '&' runs a process in the background
- Can use Control-Z to "suspend" process
  - fg: continue process in foreground
  - bg: put process in the background, e.g., to get use of terminal back

Advantages of Multithreading

- Program can do multiple things, while one task is paused or waiting on something
  - An internet browser can communicate with multiple hosts, open an email editing window, and render images
- Java uses multithreading
  - runs a garbage-collection thread in the background to deal with memory management

Concurrent vs. Parallel Programs

- Parallel program: performs more than one task on more than one processor
  - E.g., one task on each processor
- A concurrent program performs more than one task on one processor
  - Logically/usually, one task/thread
  - In reality, only one thread runs at a time
  - Each thread runs for a short period of time
  - After interval, the thread scheduler chooses another thread to run

Using Threads in Java

- Use the java.util.Timer class (and others)
- Extend the java.lang.Thread class
- Implement java.lang.Runnable interface

java.util.Timer

- Execute a task after a delay
- Execute a task at a specific time
- Each task is represented as a subclass of **TimerTask** with the **run()** method overridden
- Tasks should not take a lot of CPU time
- Timer starts on construction
  - Different from threads…

Reminder.java

- java.swing.Timer
  - Uses an ActionListener instead

Creating a Thread: Approach 1

- Create a thread by extending the **Thread** class
- A Thread class object must have a run() method
  - runs when the thread starts
  - causes the thread to exit when the run() method ends
- Example
  - a thread that will sleep (wait) for a specified amount of time
  - then wakes up and prints its info


```
class PrintThread extends Thread {
      private int sleepTime;
      public PrintThread(String name) {
            super(name);
            sleepTime = (int)Math.random() * 5000;
            System.out.println("Name:" + getName() +
                  "; Sleep:" + sleepTime);
      }
      public void run() {
            try {
                  System.out.println(getName() +
                        " going to sleep.");
                  Thread.sleep(sleepTime);
            } catch (InterruptedException exp) {
                  System.out.println(exp);
            }
            System.out.println(getName() +
                  " done sleeping.");
      }
}
```

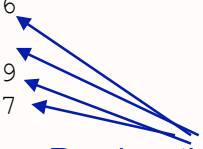
See corrected version

Then, we can make a class to use PrintThread objects…

```
public class ThreadTester {
      public static void main(String args[]) {
            PrintThread thr1, thr2, thr3, thr4;
            thr1 = new PrintThread("Thread 1");
            thr2 = new PrintThread("Thread 2");
            thr3 = new PrintThread("Thread 3");
            thr4 = new PrintThread("Thread 4");

            System.out.println("Starting Threads...");

            thr1.start();
            thr2.start();
            thr3.start();
            thr4.start();

            System.out.println("Threads started.");
      }
}
```


Output from tester class . . .

```
Name: Thread 1;  Sleep: 1446
Name: Thread 2;  Sleep: 40
Name: Thread 3;  Sleep: 1009
Name: Thread 4;  Sleep: 4997

Starting Threads
Threads Started

Thread 1 going to sleep.
Thread 3 going to sleep.
Thread 2 going to sleep.
Thread 4 going to sleep.
Thread 2 done sleeping.
Thread 3 done sleeping.
Thread 1 done sleeping.
Thread 4 done sleeping.
```



Random times--between 0 and 5 seconds

Creating a Thread: Approach 1

- A thread is created by instantiating an object of a class derived from Thread
- Thread class constructor takes one parameter: the name of the newly created thread
  - Name can be any string you want
    - Helps programmer to keep track of the threads in the program
  - Additional constructors

Creating a Thread: Approach 2

- What if want a class that is already a subclass to be a thread?
  - Implement the **Runnable** interface
  - Implement a run() method

Creating a Thread: Approach 2

- A Thread object must still be created to run the thread
  - Pass the Runnable object to the constructor for the Thread class

```
public Thread( Runnable runnableObject )
```

  - Can name the thread object as well…

```
public Thread( Runnable runnableObject,
               String threadName )
```

- Registers the run() method of the runnableObject as the code to run when this Thread object's start() method is called

Other Options

- java.swing.Timer
  - For use in GUIs
- java.swing.SwingWorker
  - Executing threads in the background

Thread Lifetime

- A thread can be in one of four states at any given moment in time
  - New
  - Runnable
  - Blocked
  - Dead

Thread States: New

- When a thread has just been created
- In this state, the program has not yet started executing the thread's internal code
- Can only call start()
  - Otherwise, get IllegalThreadStateException
- After the thread's start() method is called, it enters the runnable state.

Thread States: Runnable

- Thread is ready to execute
- Might not be actually running on the CPU
  - Operating system must schedule the thread
  - When OS schedules thread and the thread is actually running, Java still considers the thread runnable
- In most concurrent programming models, there is a distinction between the runnable and running states

Thread States: Runnable

- Each OS has policies for scheduling threads
- Most Java-supporting OSes support preemptive multithreading and time-slicing
  - allows one thread to run, then interrupting it to allow another thread to run
- Remember: a runnable thread may or may not be running at any given moment

Thread States: Blocked

- A thread enters the blocked state whenever:
  - The thread calls its sleep() method
  - The thread calls an operation that blocks on input/output
    - operation does not return from the call until the I/O operations are complete
  - The thread calls wait() method
  - The thread tries to lock an object that is currently locked by another thread
  - A different thread calls the suspend() method of the thread
    - suspend() is deprecated; don't call it
- When a thread is blocked, another thread is scheduled to run

Thread States: Blocked

- When a thread is reactivated, it returns to the runnable state
- The thread scheduler checks if the reactivated thread has a higher priority than the currently running thread
  - If thread.getPriority() > running.getPriority()
    - scheduler preempts currently running thread to schedule reactivated, higher-priority thread

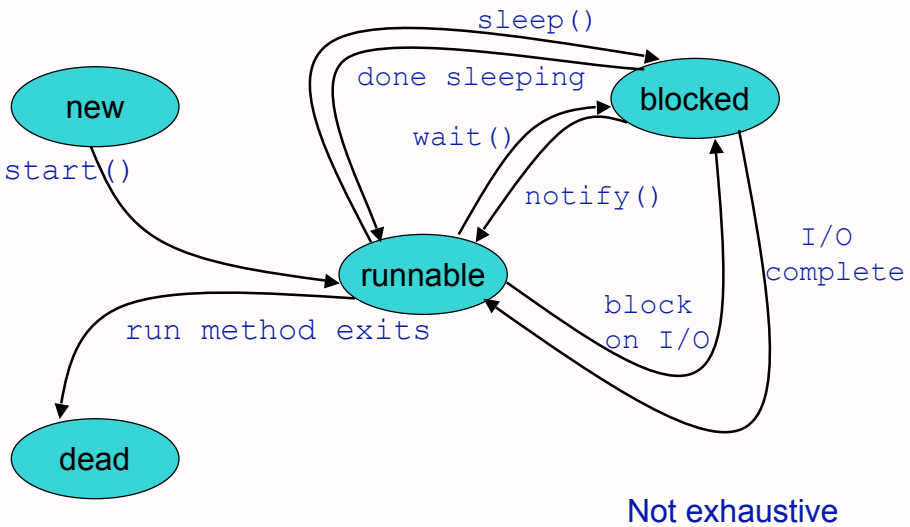
Thread States: Blocked

- A thread returns to the runnable state from the blocked state
  - If the thread called sleep(), the specified number of milliseconds to sleep has passed
  - If the thread is waiting for blocked I/O, the specified I/O request completed
  - If the thread called wait(), some other thread must call notify() or notifyAll()
  - If the thread is waiting for an object lock held by another thread, other thread must release the lock
- One-to-one correspondence between ways to enter the blocked state and ways to return from the blocked state

Thread States: Dead

- A thread can enter the dead state in one of two cases:
  - It dies naturally when the run() method exits normally
  - It dies abnormally because the run() method generated an uncaught exception
- isAlive() method
  - To find out if a thread is currently alive
  - returns true if the thread is runnable or blocked
  - returns false if the thread is new or dead

Thread States: Possible Transitions



Not exhaustive

Thread States and Methods

- Some thread methods are only valid when the thread is in certain states
  - start() can only be called on a thread in the new state
  - sleep() can only be called on a thread in the runnable state
- If a program attempts to calls a method on a thread that is in invalid state, an **IllegalThreadStateException** is thrown.

Thread States

- getState()
  - NEW
  - RUNNABLE
  - BLOCKED
    - Waiting on a monitor
  - WAITING
  - TIMED_WAITING
  - TERMINATED

Interrupting Threads

- A thread dies when its run() method ends
  - the only way a thread can end
- A thread should periodically test to see if it should terminate…

```
public void run() {
      while (no die request and still work to do)
      { do work }
// die request or no more work, so end and thread dies
}
```

Interrupting Threads

- Threads frequently sleep or wait to give other threads a chance to do work (to use the CPU)
- When a thread is sleeping, it cannot actively check to see if it should terminate
- Java provides an interrupt() method
  - When this method is called on a thread that is currently blocked, the blocking call (sleep() or wait()) is terminated with an **InterruptedException**
    - Checked exception

What does that mean?

Interrupting Threads

- Interrupting a thread using the interrupt() method does not necessarily mean that the thread should terminate
  - Grabs the attention of the thread
  - Like poking at a sleeping roommate
- Place code that will decide how to react to the interruption inside the catch clause of the run() method

The main execution loop calls sleep or wait. If the thread is interrupted during one of these calls, the catch clause runs.

```
public void run() {
      try {
            // main thread execution loop
            while (more work to do)
            {
                  do this work
            }
      }
      catch (InterruptedException exp)
      // thread was interrupted during sleep/wait
      {
            . . .
      }
// exit the run method, so the thread dies
}
```


Interrupting Threads

- If the example thread is interrupted when it is not sleeping or waiting, no InterruptedException is generated
  - the thread must check that it has not been interrupted during the main execution loop
  - interrupted()
    - returns true if the thread has been interrupted

```
while (!interrupted() && more work to do)
{   // main thread execution loop
}
```

I/O and Thread Interruptions

- If a thread is blocked waiting for I/O, the input/output operations will not be interrupted by the call to interrupt()
  - Why?
  - When the blocking I/O operation returns, the thread will check to see if it has been interrupted

Waiting for Thread Completion

- It is possible to wait for a thread to complete.
  - Call the thread's join() method
  - For instance, to wait for thread1 to complete call…

```
thread1.join();
```

- A timeout value can also be specified
  - if the thread does not die within the specified timeout, an **InterruptedException** is generated

Waiting for Thread Completion

- To wait for 500 milliseconds for thread1 to die, specify the timeout value in the call to the join() method

```
try {
      thread1.join(500);
      System.out.println("Thread exited.");
}
catch (InterruptedException exp1) {
      System.out.println("Timeout expired!");
}
```


Thread Priorities

- Each thread has a priority
  - a numerical representation of the relative "importance" of the thread running, compared to the other threads
- A thread's priority can be set using the setPriority() method
- A thread's priority must be between MIN_PRIORITY and MAX_PRIORITY
  - Constants, defined as 1 and 10
  - NORM_PRIORITY, defined as 5

Thread Priorities and Scheduling

- Fixed-priority Scheduling
  - Whenever the thread scheduler has to schedule a new thread to run, it generally picks the highest-priority runnable thread
- The highest-priority thread keeps running until either
  - it yields by calling the yield() method
  - it becomes non-runnable (dying or entering the blocked state)
  - a higher-priority thread becomes runnable
- What happens if there is more than one runnable thread with the same highest priority?

Thread Priorities and Scheduling

- What happens if there is more than one runnable thread with the same highest priority?
  - Up to the underlying operating system's policy
    - It doesn't matter which thread is chosen
  - There is no guarantee this is a fair process!
    - OS could pick a random choice or pick the first of the highest-priority threads

Program Termination and Threads

- Whenever at least one thread is alive (runnable or blocked), the program/process the thread(s) belong to is still active
- Even if the main() method exits, an alive thread will prevent a program from terminating
- A daemon thread will not prevent a program from terminating

Daemon Threads

- Daemon thread: a thread that runs solely for the benefit of other threads
  - Example: the Java garbage collector
- Daemon threads are exactly like normal threads with one difference
  - A live daemon thread will not stop the JVM from terminating
  - You need to have at least one non-daemon thread to keep the program alive
- To set a thread to be a daemon thread, call setDaemon() before calling start()
- Use isDaemon() to see if a thread is running as a daemon

Thread Groups

- To simplify working with multithreaded programs, Java introduces the concept of thread groups
- Thread Group: a convenient way of grouping related threads
- In an internet browser, multiple threads may load different images all to be displayed on the same page
  - If the user clicks the Stop button, all of these threads should be interrupted, so all of these threads are in the same thread group

Thread Groups

- Create a new thread group with a name…

```
ThreadGroup imgThrs = new ThreadGroup("ImageThreads");
```

- When a thread object is constructed, specify its group in the constructor…

```
Thread thr1 = new Thread(imgThrs, "Image Thread 1");
```

- To see if any threads in a group are runnable, call activeCount(), which will return an integer number of runnable threads in the thread group…

```
if (imgThrs.activeCount()) > 0
```

Thread Groups

- An entire group of threads can be interrupted at the same time

```
imgThrs.interrupt();
// interrupts all threads in the thread group
```

- Thread groups can have child subgroups
- A newly created thread group is, by default, a child of the current thread group
- Methods such as activeCount() and interrupt() work on all threads in their group and in all child groups

Thread Synchronization

- In most applications where multithreading is used, more than one thread will need to access the same data
  - Can turn into a problem when two threads have access to the same object and then both modify the state of the object
  - Depending on the order of data access, the object can become corrupted
  - Known as a race condition

Thread Synchronization

- Suppose we have a program that is handling transactions for a bank
- For each transfer, the program starts a separate thread to perform the transfer
- The thread gathers the necessary information and calls the transfer() method of the bank object

The transfer function deducts the transfer amount from the indicated account and adds it to the indicated account.

```
class Bank {
      int num_transfers;
      . . .
      public void transfer(int from, int to,
                           int amount)
      {
            if (accounts[from] < amount)
                  return;
            accounts[from] -= amount;
            accounts[to]   += amount;
            num_transfers++;
      }
}
```

Thread Synchronization

- Suppose we have two transfer threads that run at once
  - Remember that a thread can be interrupted at any time
- Note that the statement

```
accounts[from] -= amount;
```

is really composed of three parts

(1) loading accounts[from] into a register

(2) subtracting amount from it

(3) saving the result back into memory

Thread Synchronization

- The program has two transfer threads running and they both call the transfer function of the Bank object, each of which will execute these three steps
- Where is the potential problem?

Thread 1:

```
LOAD   ACCOUNTS[from], R1
ADD    R1, amount
SAVE   R1, ACCOUNTS[from]
```

Thread 2:

```
LOAD   ACCOUNTS[from], R1
ADD    R1, amount
SAVE   R1, ACCOUNTS[from]
```

Thread Synchronization

- Since a thread can be interrupted at any time for another thread to run, this order of execution is possible…

```
THR1: LOAD   ACCOUNTS[from], R1
THR1: ADD    R1, amount

THR2: LOAD   ACCOUNTS[from], R1
THR2: ADD    R1, amount
THR2: SAVE   R1, ACCOUNTS[from]

THR1: SAVE   R1, ACCOUNTS[from]
```

Thread 1 gets preempted before it would have written its new data out

Thread 2 runs and updates this value

Thread 1 runs again and overwrites Thread 2's changes!

Atomic Operations

- The incorrect behavior is possible because the statement that subtracts the amount from the account (and the one that adds the amount to the account) is not an atomic operation.
- Atomic operation: an operation that is guaranteed to run to completion once it has begun
  - The subtract and add operations can be interrupted after they started and before they complete

Atomic Operations

- If the transfer method was guaranteed to run to completion once it was entered, this problem could not happen
- Traditionally, atomic operations are implemented using mutexes, semaphores, and critical sections
  - More in your OS class!
- Java introduces a monitor, a simpler way of handling atomic operations

Synchronized Methods

- A method or block of code can be marked as atomic
  - Using keyword **synchronized** in method's return type
  - After the section of code starts, the code is guaranteed to complete before another thread can enter another synchronized section belonging to that same object

The transfer function is atomic (synchronized). After it begins, it cannot be preempted until it completes.

```
class Bank {
      int num_transfers;
      . . .
      public synchronized void transfer
            (int from, int to, int amount)
      {
            if (accounts[from] < amount) return;
            accounts[from] -= amount;
            accounts[to]   += amount;
            num_transfers++;
      }
}
```


Synchronized Methods & Objects

- So, what does making a method synchronized do?
- When a thread calls a synchronized method of an object, that object becomes locked
  - There is exactly one shared key or lock for an object
  - To enter any synchronized method of that object, you must have the key
  - When you have the key, you unlock the door to the synchronized method you want to enter and take the key with you
  - When another thread attempts to enter a synchronized method, it cannot find the key, so it blocks
    - Other thread waits on the doorstep to the object for the key to return

Synchronized Methods & Objects

- When you return from your synchronized method, you leave the object's key behind on the doorstep.
- Other threads can retrieve the key and enter one of the object's synchronized methods

- When a thread calls an object's synchronized method, the object (not the specific method) is locked
- Only one key for each object
  - Key is needed to enter any synchronized method
  - Key represents the lock in other paradigms
  - If a synchronized method is entered, no other synchronized method of that object can be entered.

Synchronized Methods & Objects

- Other threads are can call and execute unsynchronized (normal) methods of that object
  - only the synchronized methods are blocked
- If a thread owns the lock on an object, it can call another synchronized method of the same object
  - The thread releases the lock only after exiting the last synchronized method
- A thread can own the lock on more than one object at a time
  - call a synchronized method of one object from inside a synchronized method of another object

One Additional Problem

- The program should not transfer money out of an account that does not have the funds to cover it
- We need to check before we start the transfer.

- What is wrong with this approach?

```
if (bank.getBalance(from) >= amount)
     bank.transfer(from, to, amount);
```


One Additional Problem

```
if (bank.getBalance(from) >= amount)
     bank.transfer(from, to, amount);
```

- This thread could get preempted after the test and before the call to transfer()
- The newly running thread could transfer money out of the "from" account, leaving insufficient funds.
- Solution: Move the test inside the synchronized transfer() method

One Additional Problem

```
public synchronized void transfer(int from,
            int to, int amt) {
      while (accounts[from] < amount)
            { wait }
      // then, transfer funds
}
```

- A thread cannot be preempted between the test and transfer
- What should the thread do when there are not enough funds?
  - The thread could wait until another thread has added the necessary funds
  - What's the problem with this approach?

The wait() Method

- The transfer() method is synchronized!
- No other thread can run since the current thread is in the synchronized method and waiting
- There is a feature of synchronized methods which takes care of this situation
  - A thread can call the object's wait() method to wait inside of a synchronized method
  - The current (calling) thread blocks
  - The object's lock is released

wait() and the Wait List

- After a thread calls wait(), it is added to a wait list
  - List of threads waiting on a certain object
- The thread scheduler ignores threads on the wait list
  - Waiting thread does not continue running
- To remove the thread from the wait list, some other thread must call the object's notify() or notifyAll() method
- notify() removes one arbitrary thread from the waiting list
- notifyAll() removes all waiting threads

The wait()/notify() Mechanism

- After a thread has been removed from the wait list, the scheduler will eventually run it again
  - The thread will attempt to lock the object because it gave up the lock when it called wait()
- After the thread gets the lock, it reenters the object and continues where it left off
  - after the call to wait()
  - known as reentry

The wait()/notify() Mechanism

- Other threads **must** call notify() or notifyAll()
  - Otherwise, get deadlock
- When a thread calls wait(), there is no way of unblocking it unless another thread calls notify()
- The waiting threads are not automatically reactivated when no other thread has the object locked
- If all threads but one are blocked and the unblocked thread calls wait(), the program will wait forever because there is no thread left to call notify()

notify() vs. notifyAll()

- It is somewhat confusing as to when a program should call notify()
  - no control about which thread will become unblocked
- Generally, call notifyAll() whenever the state of the object changes in such a way that might be advantageous to other threads
  - Will need to handle that the condition that caused to block still isn't met
- If the account balance of an account changes, multiple threads may be blocked based on this event occurring
  - wake them all up

The wait()/notify() Mechanism

- Calling notifyAll() does not immediately activate all of the waiting threads
  - It unblocks the waiting threads and removes them from the wait list
  - The threads still must compete for entry into the object after the current thread has exited its synchronized method

Synchronization Review

- To execute a synchronized method, the object must not be locked
  - The calling object acquires the lock
- Returning from a synchronized method releases the object's lock.
- Only one thread can execute synchronized methods of a particular object at one time
- When a thread calls wait(), it releases the lock on the object and enters a wait list
- To remove a thread from the wait list, another thread must call notify() or notifyAll() on the object

Writing Multithreaded Code

- If two or more threads modify an object, declare the methods that carry out these modifications as synchronized
  - Basically, putting a lock on the code
- If a thread must wait for the state of an object to change, it should wait *inside* the object by entering the appropriate synchronized method and calling wait().

Writing Multithreaded Code

- Whenever a method changes the state of an object, it should call notifyAll()
  - Gives waiting threads a chance to see if the situation that caused them to wait has changed, allowing them to continue
- wait() and notify()/notifyAll() are methods of the **object**, not the thread
- All calls to wait() are matched up with a notification call on the same object

Deadlocks

- **Deadlock**: when threads are stuck, waiting for another to do something first
  - A deadlock could involve a ring of threads, waiting for each other or simply one thread waiting for itself

Example of Deadlocking

- Suppose the bank has two accounts, account #1 has $5000 in it and account #2 has $3750 in it.
- Thread #1 wants to transfer $4000 from account #2 to account #1.
- Thread #2 wants to transfer $6000 from account #1 to account #2.
- These two threads are deadlocked, as neither can run until the other one has finished.

Avoiding Deadlock

- There is no support in Java for avoiding or preventing deadlock situations
- Programmer must be very careful not to write programs in which deadlock can occur
  - Careful synchronization: keep it simple

Starvation

- One thread never gets time on the CPU
- To avoid starvation
  - May need higher-priority threads to call sleep or yield periodically

Threads in GUIs

- Recall ColorChooserDemo.java from 7/11?
- Swing isn't thread-safe
  - Very few methods in Swing are synchronized
  - If multiple threads update the model, may not see correct GUI
- Swing works in the **event dispatch thread**
  - Handles event handling, repainting
  - Each event will finish before next
  - Repainting is not interruptable

Threads in GUIs

- public static void invokeLater(Runnable r)
  - Adds the specified Runnable object to the event queue
  - Returns immediately

Rules of GUI Programming with Threads

- If an action takes a long time, fire up a new thread to do the work. Otherwise, the GUI will appear to be dead until the work is complete.
- If an action might block on IO, perform that action in a new thread.
- If you need to wait for a specific amount of time, don't sleep in the event dispatch thread
  - Instead, use a Timer.
- One thread to rule them all (single thread rule)
  - The work that you do in your threads cannot touch the user interface
  - Update the GUI from within the event dispatch thread after the GUI is displayed

Other Java Classes for Concurrency

- Package java.util.concurrent.locks
  - Lock
  - Condition
  - Match more traditional OS synchronization programming
- Synchronized data structure classes in java.util.concurrent
  - Hide synchronization details
- java.util.concurrent.atomic
  - Allow thread-safe updates of objects