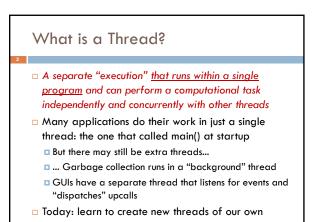


## THREADS AND CONCURRENCY

Lecture 22 – CS2110 – Spring 2013



## What is a Thread?

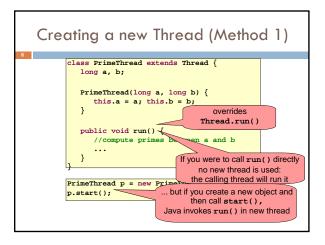
- A thread is a kind of object that "independently computes"
  - Needs to be created, like any object
  - Then "started". This causes some method (like main()) to be invoked. It runs side by side with other thread in the same program and they see the same global data
- The actual execution could occur on distinct CPU cores, but doesn't need to
  - We can also simulate threads by multiplexing a smaller number of cores over a larger number of threads

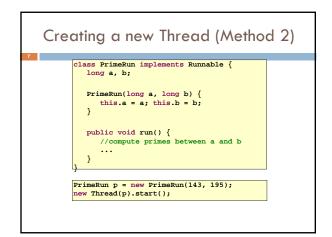
## Concurrency

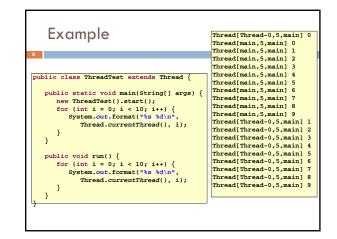
- Concurrency refers to a single program in which several threads are running simultaneously
  - Special problems arise
- They see the same data and hence can interfere with each other, e.g. if one thread is modifying a complex structure like a heap while another is trying to read it
- □ In cs2110 we focus on two main issues:
  - Race conditions
  - Deadlock

## Thread class in Java

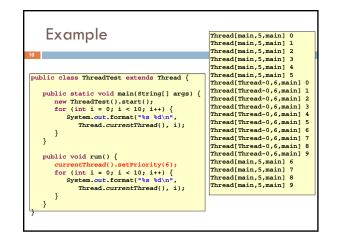
- Threads are instances of the class Thread
- Can create many, but they do consume space & time
- The Java Virtual Machine created the thread that executes your main method.
- Threads have a priority
  - Higher priority threads are executed preferentially
     A newly created Thread has initial priority equal to the thread that created it (but can change)

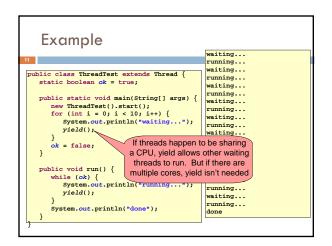


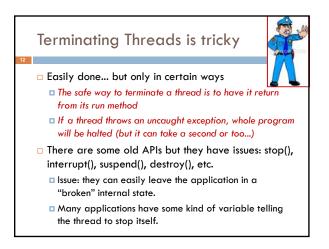




<pre>public class ThreadTest extends Thread {     public static void main(String[] args) { </pre>	Thread[main,5,main] 0 Thread[main,5,main] 1 Thread[main,5,main] 2 Thread[main,5,main] 4 Thread[main,5,main] 4 Thread[main,5,main] 6 Thread[main,5,main] 7
<pre>new ThreadTest().start();</pre>	Thread[main,5,main] 8
for (int i = 0; i < 10; i++) {	Thread[main,5,main] 9
System.out.format("%s %d\n",	Thread[Thread-0,4,main] 0
Thread.currentThread(), i);	Thread[Thread-0,4,main] 1
}	Thread[Thread-0,4,main] 1
public void run() {	Thread[Thread-0,4,main] 3
currentThread().setPriority(4);	Thread[Thread-0,4,main] 4
for (int i = 0; i < 10; i++) {	Thread[Thread-0,4,main] 5
System.out.format("%s %d\n",	Thread[Thread-0,4,main] 6
Thread.currentThread(), i);	Thread[Thread-0,4,main] 8
}	Thread[Thread-0,4,main] 8
}	Thread[Thread-0,4,main] 9







#### Threads can pause



- When active, a thread is "runnable".
  - It may not actually be "running". For that, a CPU must schedule it. Higher priority threads could run first.
- A thread can also pause
  - $\hfill\square$  It can call Thread.sleep(k) to sleep for k milliseconds
  - If it tries to do "I/O" (e.g. read a file, wait for mouse input, even open a file) this can cause it to pause
  - Java has a form of locks associated with objects.
     When threads lock an object, one succeeds at a time.

## Background (daemon) Threads 🕻



- In many applications we have a notion of "foreground" and "background" (daemon) threads
  - Foreground threads are the ones doing visible work, like interacting with the user or updating the display
  - Background threads do things like maintaining data structures (rebalancing trees, garbage collection, etc)
- On your computer, the same notion of background workers explains why so many things are always running in the task manager.

### **Race Conditions**



- A "race condition" arises if two or more threads access the same variables or objects concurrently and at least one does updates
- Example: Suppose t1 and t2 simulatenously execute the statement x = x + 1; for some static global x.
   Internally, this involves loading x, adding 1, storing x
   If t1 and t2 do this concurrently, we execute the
  - statement twice, but x may only be incremented once t1 and t2 "race" to do the update

Race Conditions		
<ul> <li>Suppose X is initially 5</li> </ul>		
Thread t1	Thread t2	
LOAD X	□	
DADD 1	LOAD X  ADD 1  STORE X	
I after finishing, X=6! We "lost" an update		

#### Settings where race conditions matter

- Two or more threads try to access something, and one or more want to change it
- A for-each loop is iterating over a collection, but some thread modifies the collection concurrently
- You want your program to do one thing at a time, e.g. so that the user can fill in a form without being interrupted to fill in some other form
- ... this list is very partial!

## **Race Conditions**

Race conditions are bad news

- Sometimes you can make code behave correctly despite race conditions, but more often they cause bugs
- And they can cause many kinds of bugs, not just the example we see here!
- A common cause for "blue screens", null pointer exceptions, damaged data structures

## Example – A Lucky Scenario

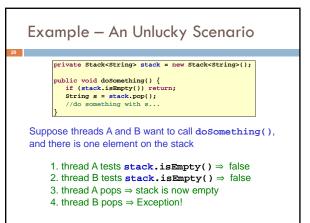
private Stack<String> stack = new Stack<String>();

public void doSomething() {
 if (stack.isEmpty()) return;

String s = stack.pop();
//do something with s...

Suppose threads A and B want to call dosomething(), and there is one element on the stack

- 1. thread A tests stack.isEmpty() false
- 2. thread A pops  $\Rightarrow$  stack is now empty
- 3. thread B tests **stack.isEmpty()**  $\Rightarrow$  true
- 4. thread B just returns nothing to do

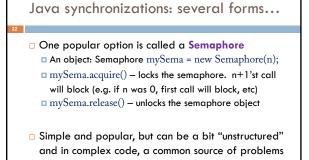


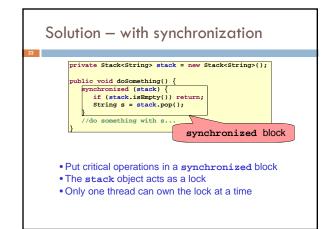
# Synchronization

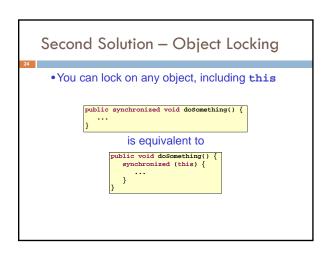
 Java has one "primary" tool for preventing these problems, and you must use it by carefully and explicitly – it isn't automatic.

Called a "synchronization barrier"

- We think of it as a kind of lock
  - Even if several threads try to acquire the lock at once, only one can succeed at a time, while others wait
  - When it releases the lock, the next thread can acquire it
  - You can't predict the order in which contending threads will get the lock but it should be "fair" if priorities are the same







## Synchronization+priorities

- Combining mundane features can get you in trouble
- Java has priorities... and synchronization
  - But they don't "mix" nicely
  - High-priority runs before low priority
  - ... The lower priority thread "starves"



- Even worse...
  - With many threads, you could have a second high priority thread stuck waiting on that starving low priority thread! Now <u>both</u> are starving...



- Java developers have created various synchronization ADTs
  - Semaphores: a kind of synchronized counter
  - Event-driven synchronization
- The Windows and Linux and Apple O/S all have kernel locking features, like file locking
- □ But for Java, synchronized is the core mechanism

## Deadlock



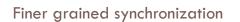
□ The downside of locking – deadlock

A deadlock occurs when two or more competing threads are waiting for one-another... forever

#### Example:

Thread t1 calls synchronized b inside synchronized a

- But thread t2 calls synchronized a inside synchronized b
- t1 waits for t2... and t2 waits for t1...



Java allows you to do fancier synchronization
 But can only be used <u>inside</u> a synchronization block
 Special primatives called wait/notify

