The UNIVERSITY of VERMONT

Benjamin Fenster
CS-100 (Damon)
6 April 2003

Notes - Threads

I. Flow of Control
   a. What happens next?
   b. So far everything has been a single flow of control
      i. There's only one answer to the question "Where is it now?"
      ii. Only one thing happening at a time.
   c. Some programs need multiple threads of control
      i. "Multiple Threads"
      ii. This is particularly useful when you want one thing to execute without waiting for another.
      iii. User interfaces often want to be in a separate thread so the interface will continue reacting even when the program is doing other stuff in the background.
      iv. Use whenever you want to run a background task
      v. Use whenever you don't want the program to wait for something.
   d. Some processes are easy to break into small tasks
      i. If so, do a little piece, then check the UI again.
      ii. Do a "round robin" of tasks.
      iii. No need for multiple threads.
   e. Other tasks just take a long time, and can't easily be broken apart
      i. Go do your piece and come back when you're done.
      ii. I'll do other stuff while I wait.

II. Threads
   a. Built from Runnable
   b. One method, run(), which does the work.
   c. The thread ends when run() ends.
   d. Think of it like a main() for a whole new program
   e. run() must be completely independent in terms of execution.
   f. Thread implements Runnable
   g. Control and Manipulation
      i. `Thread.sleep(long milliseconds)`. Just don't do anything for some length of time.
      ii. `static Thread.currentThread()` gives the thread from which the call is made.

III. Flow of Control
   a. Running threads can change the order in which code is executed.
   b. Within any one thread, the order is "correct" (as expected)
   c. Remember that statement interleaving is at the machine instruction level, not the "line of code" level.
   d. Among several threads though, there's no way to tell which is running a chunk of code at any given moment.
   e. If the threads are completely independent, nobody cares.
   f. Remember that all threads see the same memory though, so if they are trying to read or change the same object there could be a problem.
      i. Could have two pieces of code retrieve the same value, manipulate it differently, and store the wrong result.
      ii. See CS100-29-13
      iii. Almost all problems come back to violations of the rep invariant.
      iv. You cannot assume the invariant is true in the middle of a method, but it's assumed to be true at the beginning.
      v. With methods running in the middle of other methods, that can cause problems.
   g. Synchronized
      i. Declare methods as synchronized

ii. No other synchronized method can run until the first finishes.
iii. There's still no way to know which synchronized methods will be run first, but the statements will no longer be interleaved.
iv. It doesn't matter what CODE is running (the same method can even be running more than once), it matters what object is being accessed.
v. Synchronized methods on the same object are off-limits.

h. synchronized statement
   i. It's not always necessary to synchronize an entire method.
   ii. `synchronized (objectExpression) { code; }`
   iii. That blocks all methods and statements that are synchronized and works exactly like it did for methods.

IV. The Problem
   a. What if you need two different objects, each in two different threads.
   b. Each thread may get one object, and neither can do anything.
   c. Called a deadlock.
   d. Always avoid using multiple locks whenever possible.
   e. Also always try to grab things in the same order.
   f. It's still really hard to guarantee a deadlock will never happen.