Slide 1

Administrivia

- Homework 2, both parts, now due next Monday.

Slide 2

Multithreaded Programming in Java — Overview

- We'll look next at basic multithreaded programming in Java, mostly focusing on lower-level approaches.
- Why Java? When we wrote the book it was a language many people already knew, including Trinity CS students (after CS2). Now still popular but not used in our required courses. So why . . .

Slide 3

Java in This Course

- Students no longer come into this course knowing some Java, but I say there's reason to use it anyway:
- Some examples make more sense if you know a little Java.
- Even limited exposure to another popular language could be an asset.
- Am I trying to teach you Java? not really — "just enough to be dangerous", i.e., enough to understand examples and to adapt starter/example code.
- Dr. Lewis has a partial set of video lectures ("reading" for today), which you may find helpful.

Slide 4

Introduction to Java for People Who Know Scala and C

- Scala is built on top of Java and shares a common runtime environment. Conceptually similar in many way.
- Syntactically, however, Java is more like C in many ways. (Once I could say "more like C++", but C++ is evolving . . .)
- And then the language just has some quirks all its own . . .

Slide 5

How Java is Like Scala

- Designed to protect programmers from themselves — memory management via garbage collection, many errors throw exceptions.
- *Huge* standard library.
- Support for object-oriented programming. Prior to Java 8, however, support for functional programming imperfect at best. (And I won't discuss Java 8.)

Slide 6

How Java is Like C

- No REPL environment (alas). Must compile (to "byte code") and then execute using runtime system.
- Programs all have to include some "boilerplate" lines that set things up for the main program.
- Most syntax closer to C than Scala. Statements must end with semicolons.
- Must specify a type when declaring variables; no `val` versus `var` distinction (but there is `final`).

Slide 7

Java and Object-Oriented Programming

- Basic object-oriented ideas (classes and inheritance) mostly the same in Java as in C++ and Scala; syntax and details are more similar to C++ than Scala.
- Classes can include both regular and `static` members. (`static` members akin to Scala companion objects.) Members include data, methods, and (nested/inner) classes.
- Various "access modifiers" (`public`, `private`, etc.) limit accessibility of classes and their members.
- Classes can be grouped into packages, as in Scala.

Slide 8

Java and Object-Oriented Programming, Continued

- Type-generic programming possible; syntax more like C++ than Scala.
- No multiple inheritance. C++ has this. Scala doesn't, but allows "traits". Java has "interfaces", akin to limited version of Scala traits. (Traits can have variables, method code; interfaces are purely an API — method declarations — plus maybe constants.)

Slide 9

Java Peculiarities

- Everything in Java is part of some class; no free-standing functions.
- Variable types include "primitives" (lowercase type name, similar to C variables) and "references" (uppercase type name corresponding to a class, similar to Scala variables). Why oh why? Attempt at efficiency.
- No function pointers, and prior to Java 8, no support for functions as first-class objects or "lambda expressions". Workaround is to use interfaces and (sometimes) "anonymous classes". Example when we talk about defining threads shortly.

Slide 10

Java Peculiarities, Continued

- Exception mechanism similar to the one for Scala, *but*:

  "Unchecked" exceptions can be caught, or not, as you choose. For "checked" exceptions, however, must either catch them or explicitly declare that your method can throw them. Meant to be a good thing — forcing you to think about exceptions that are common enough that you shouldn't just pretend they can't happen — though in practice sometimes annoying.
- Compiler picky about names: Only one public class per file, and name of file must match name of class.
- Runtime system picky about directory structure; must match package structure.

Slide 11

"Hello World" in Java

- Define class `Hello` in file `Hello.java`.
- (You can use Eclipse for Java, but for short programs I don't, and sometimes (especially for this class) it's better to run from the command line. So I'll show command-line tools only. If you really really like Eclipse you might consider using it to write code but then executing from command line.)
- Compile with `javac Hello.java`. If it succeeds, generates a file `Hello.class`. (To reduce clutter, add `-d objectdir`.)
- Execute with `java Hello`. (If you compiled with `-d`, add `-cp objectdir`.)

Slide 12

Parallel Programming in Java

- Java supports multithreaded (shared-memory parallel) programming as part of the language — `synchronized` keyword, `wait` and `notify` methods of `Object` class, `Thread` class. Programs that use GUI libraries multithreaded under the hood. (Scala shares this property.) Justification probably has more to do with hiding latency than HPC, but still useful, and versions 5.0 and beyond includes much useful library stuff.
- Java also provides support for forms of distributed-memory programming, through library classes for networking, I/O (`java.nio`), and Remote Method Invocation (RMI).

Slide 13

What Does A Multithreaded Java Program Look Like?

- Easy answer: Like a regular Java program.
- Programming model: All threads share a common address space. Programmer is responsible for creating threads, providing synchronization, etc.

Slide 14

Creating Threads in Java

- Threads are all instances of `Thread` class (or a subclass). Pre-5.0, two ways to create threads:
  - Create a subclass of `Thread` (frowned on by o-o purists).
  - Create a `Thread` using an object that implements `Runnable` interface (preferable).

  Either way, `run` method (of subclass of `Thread`, or of `Runnable`) contains code for thread to execute.
- Start thread with `start` method. Can wait for it to finish with `join`.
- "Hello world" example (`Hello1.java` and `Hello2.java` on sample programs page). (Other methods in `java.util.concurrent` — see sample programs `Hello3.java`, `Hello4.java`, `Hello5.java`.)

Slide 15

Shared Variables in Java

- Code executed by a thread is some object's `run` method. Access to variables consistent with usual Java scoping — class/instance variables, parameters, etc.
- As we noted before, though, simultaneous access to shared variables can be risky, however. So . . .

Slide 16

Synchronization in Java

- Interaction among threads in Java based on "monitor" idea (Hoare (1975) and Brinch Hansen (1975)).
- Every object has implicit lock; `synchronized` keyword means "only run this when you have the relevant lock" — if another thread has the lock, wait. Can be used to ensure one-at-a-time access to critical variables.

  "Relevant lock"? For synchronized methods, lock for object (instance methods) or class (static methods). For synchronized blocks, you specify the object.

  Example — `HelloSynch*.java` on sample programs page.
- `wait` and `notify` methods allow more interesting kinds of coordination. But first . . .

Slide 17

Numerical Integration Example, Revisited

- How to parallelize using Java? well, first must rewrite in Java (`NumIntSeq.java` on sample programs page).
- Now rewrite to use multiple threads, based on same strategy we used for OpenMP — split loop iterations among threads, give each its own copy of work variables, compute sum based on "reduction" idea. Some things must be done more explicitly in Java (making the program in some ways more like MPI's SPMD model); see `NumIntPar1.java` on sample programs page.

  Notice however that this problem would make good use of `java.util.concurrent`'s support for tasks/threads; see `NumIntPar2.java` on sample programs page.

Slide 18

Synchronization in Java, Continued

- `synchronized` methods/blocks can be used to ensure that only one thread at a time accesses some shared variable.
- For more complex synchronization problems, can use `wait` and `notify` (or `notifyAll`):

  `wait` suspends executing thread (adds to "wait set").

  `notify` wakes up one thread from the wait set. `notifyAll` wakes up all threads in the wait set. Newly-awakened thread(s) then compete to reacquire lock and continue execution.

  Can only be done from within synchronized method/block.

  Typical idiom — loop to check condition, `wait`.
- (More about this, and example, later.)

Slide 19

Minute Essay

- Do you have previous experience with Java?
- Did this one more version of the numerical integration example make sense?
- Questions?