Slide 1

Administrivia

- Reminder: Homework 1 MPI program due today (11:59pm).
- About reading for today: Dr. Lewis's video lectures (Java for Scala programmers) are a good start, but incomplete.

Slide 2

Java for Scala Programmers, Continued

- Scala uses Java's "infrastructure" (byte code and JVM), and you've probably already used, from Scala, some of Java's library classes. Scala's notion of packages also comes from Java.

  A notable difference is that the Java compiler is much pickier about names of packages matching the filesystem hierarchy.
- Conceptually and with regard to syntax, however, Java may be closer to C++.

Slide 3

Java for Scala Programmers, Continued

- Variables can be “primitives” (similar to variables in C) or (references to) objects. Variables declarations must include types. No `var` or `val` modifiers, but `final` has much the same effect as `val`.
- As in Scala, parameter passing is strictly by value, but if what’s passed is a reference, called method can modify the referenced object. Nothing like that is possible with primitive types.

Slide 4

Java for Scala Programmers, Continued

- Basic object-oriented ideas (classes and inheritance) basically 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. Members include data, methods, and (nested/inner) classes.
- Various “access modifiers” (`public`, `private`, etc.) limit accessibility of classes and their members.
- Type-generic programming possible with “generics”. Basically the same as in Scala, but syntax is different.

Slide 5

Java for Scala Programmers, Continued

- The exception mechanism for Java is similar to the one for Scala, with one notable exception:
- “Unchecked” exceptions can be caught, or not, as you choose. For “checked” exceptions, however, you must either catch them or explicitly declare that your method can throw them. This was 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 it can be annoying.

Slide 6

Java for Scala Programmers, Continued

- No multiple inheritance — inventors felt that that was more a source of potential confusion than a help. Instead Java has “interfaces” — somewhat similar to Scala traits, but without the ability to define variables and (definitions of) methods. I.e., an interface is basically an API only — list of methods plus possibly some constant values. A class can only inherit from one superclass, but it can implement multiple interfaces.
- No function pointers, and prior to Java 8, no lambda expressions. In situations needing either one, typical approach is to use interfaces and “anonymous classes”.

Slide 7

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 the GUI classes (AWT or Swing) are 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 8

What Does A Multithreaded Java Program Look Like?

- Easy answer: Like a regular Java program. (In fact, any program with a GUI . . .)
- Programming model: All threads share a common address space. Programmer is responsible for creating threads, providing synchronization, etc.

Slide 9

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` (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 10

Java from the Command Line

- Most of you probably use Eclipse to write Scala programs. This works for Java programs too (in fact Eclipse was largely developed as a tool for Java), but I say best to run them from the command line, where it’s easy to vary environment variables and command-line arguments. Command to use is `java`, followed by class name and any arguments. (If class files are not in current directory, specify where they are with `-classpath` or `-cp`.)
- You can also write them using your favorite text editor compile from the command line. Command to compile is `javac`. Use `-d` to put byte-code files in separate directory (probably a good idea) and `-cp` if what you’re compiling uses your own library code.

Slide 11

Shared Variables in Java

- Code executed by a thread is some object's `run` method. Access to variables is 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 12

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 13

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 14

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.)

Minute Essay

- Anything interesting to report about the MPI of Homework 1? no need to repeat (in detail anyway) what you said in the discussion you turned in with your code, but just figuring out how to run MPI programs can be a bit of a hurdle?