Slide 1

Administrivia

- Homework 6 deadline extended to Thursday.

Slide 2

Loops — Review

- `for` loops (really "for comprehensions") useful when how many times you want the loop to execute is known — e.g., depends on the size of an array or list. (See loop versions of array/list demo programs for examples. Note that while the array program uses a range of indices and the list doesn't, either way works with both arrays and lists. Style issue, maybe.)
- `while` and `do while` loops useful when you don't necessarily know ahead of time how many times — e.g., keep going until input is "quit".
- Both kinds are apt to make use of `var` variables. Notice that it is perfectly okay, for this kind of variable, to calculate and store a new value based on the current value, e.g.,

  ```
  var n = 0
  ```

  and later

  ```
  n = n + 1
  ```

Slide 3

Loop Examples, Continued

- (Finish/improve "checkbook balance" program frm last time.)
- Here too redirection might be useful — e.g., put a list of commands in `input.txt` (using a text editor!) and then run using

  ```
  scala balance-plus.scala < input.txt
  ```

  or the somewhat UNIX-geek-y

  ```
  cat input.txt - | scala balance-plus.scala
  ```

  to first process the input in `input.txt` and then accept interactive input.

  But really this is a good motivating example for the next topic (files).

Slide 4

Sidebar: Multidimensional Arrays

- Arrays and lists are useful for representing the kinds of things we might use subscripted variables in math. What about variables with multiple subscripts, though? such as matrices?
- Like many programming languages, Scala provides "multidimensional arrays" for this purpose. Like Java (on which it is based), Scala represents these as arrays of arrays — which makes a kind of sense, no?
- (For now we will skip examples of using these. See textbook.)

Slide 5

Files — Overview

- One of the things that's useful about computers is their ability to store large amounts of information in a form that they can process — i.e., the ability to store and work with *files*.
- "File" is a pretty broad and generic term and includes everything from simple text files (such as the ones that contain your Scala programs) to word-processing documents and images and digital representations of music and video and . . .
- Up to now, our ability to work with files has been limited to what I/O redirection provides — which is useful, but very limited since we can only work with one source and one destination. Most programming languages provide something more general.

Slide 6

Sidebar: Packages in Scala

- Before talking about working with files in Scala, useful to know a little about *packages*.
- Basic idea of packages is to provide some way to organize lots and lots of code: Languages may include extensive libraries. Real applications typically involve quite a lot of code. How to organize? One way is to somehow group related functionality. Scala (and Java) does this using packages. Idea is similar to folders/directories for organizing files.
- Packages also provide a nice mechanism for avoiding naming collisions — i.e., names of things (such as `List`) don't have to be unique across everything in the library and your own code, only within a package.

Slide 7

Sidebar: Packages in Scala, Continued

- You may notice that when you type an expression into the interpreter, it tells you its type, and sometimes the type is something simple (e.g., `Int`) but sometimes it's less scrutable — e.g., for a range (such as `0 to 5`) it's `scala.collection.immutable.Range.Inclusive`

  The lower-case parts identify the "package" containing the library code for ranges.

  You could use this whole name as the type for a function parameter, but that's unwieldy, so . . .
- `import` gives you a way to tell the Scala compiler/interpreter where to look for things it couldn't otherwise find. (The above isn't the best example because everything in `scala.collection` is automatically imported.)

Slide 8

Files in Scala

- Simplest way to read files in Scala is with `scala.io.Source` (or just `Source` with an `import scala.io.Source`):

  ```
  Source.fromFile("somefile")
  ```
- This gives you back something that the interpreter claims is an "iterator". What's that . . .

Slide 9

Sidebar: Iterators

- For arrays and lists you know it's sometimes useful to be able to go through every element of the array/list and do something (print it, or add it to a running total, e.g.). It's useful to be able to do that with other kinds of collections too, (e.g., lines in a file).
- Abstract term for something that lets you "visit" each element of a collection — *iterator*. As used in Scala/Java, it's something with two operations, "is there another element?" and "give me the next element".
- Something to know about iterators — not necessarily reusable (so must be somehow reset or recreated if you want to go through the collection more than once).

Slide 10

Files in Scala, Continued

- What you get back from `fromFile` is an iterator over the characters of the file, and you can apply to it lots of the methods you use on arrays and lists.
- To read a line at a time — `getLines`, which gives you an iterator over `String`s.
- After using a file, good practice to "close" it (free up any resources used to manage it).

Slide 11

Files in Scala, Continued

- Other ways of working with input files, and all ways of working with output files, use the underlying Java libraries.
- Two simple ones — `Scanner` for input, `PrintWriter` for output. Simple example:

```
import java.io.File
import java.io.PrintWriter
val pw = new PrintWriter(new File("out.txt"))
pw.println("hello world")
pw.close
```

(More examples in textbook.)

Slide 12

Minute Essay

- We looked at a few things you can do with files, some of them maybe interesting. Can you think of other programs you could now write that might be interesting?