Slide 1

Administrivia

- Okay to skim long examples in chapters 9 and 10.
- Homework 5 on the Web. Due Tuesday after the holiday.
- I plan to have class Tuesday, but I don't plan to cover essential material — longer / more-interesting(?) examples probably.

Slide 2

A Little About Errors in Scala (Exceptions)

- You've noticed that some kinds of user-input error result in rather ugly program crashes. If you wonder why, and what to do about it . . .
- The "why": If writing programs meant to teach you about programming, has its advantages, as compared to other languages, which sometimes allow you to continue even when it wouldn't make sense to do so. (Think about what happens if you prompt the human for an integer and he/she enters something else.)
- The "how": Crashes come from Scala's main mechanism ("exceptions") for signalling that something has gone wrong.

  A short discussion of errors in general, then . . .

Slide 3

Errors in Programs

- Some errors in programs are caused by programmer mistakes — e.g., trying to access an element of an array using an index that's out of bounds (in Scala, negative, or more than the array size minus 1).
- Other errors have external causes — e.g., input that's not what was intended, or files that can't be found.
- What to do about errors is something to decide when designing a programming language.
- Different languages use different approaches:

  Some try to detect and warn about programmer mistakes (safer but possibly inefficient); others don't.

  All(?) try to detect and do something about external-causes mistakes, but what they do varies.

Slide 4

Errors in Programs, Continued

- Some example errors:
  - Accessing an array element with an out-of-bounds index.
  - Getting square root of a negative number, with a function that returns a `Double`.
  - Converting text to an `Int`, when the text doesn't represent an integer.
  - Opening a file when there is no such file.
- Some things the programming language could do:
  - Ignore the error (only for programmer mistakes).
  - Return a "didn't work" value (not always possible).
  - Set a global variable somewhere (ugly).
  - Bail out of normal program control flow via *exception*.

Slide 5

A Little (More) About Errors in Scala

- Scala uses exceptions to signal most kinds of errors, including some programmer mistakes. When some kinds of errors are detected, the code that detects them (e.g., `toInt` or `fromFile`) "throws an exception".
- By default, this causes the program to crash, with error messages that are meant to be helpful to the programmer — but probably will baffle or annoy an end user.
- If you want some other behavior, you can "catch the exception".

  A very bad idea for programmer mistakes; a very *good* idea for other errors. (And I usually put a lot more emphasis on this, but this semester with this textbook I haven't.)

  Details in POP II, but for now a short example . . .

Slide 6

Case Classes — Motivation

- Arrays, lists, and loops were all introduced with the comment that even without them you can compute anything computable, but all of these constructs make it easier to do some things, or to do them in a way that may be easier for to understand.
- Case classes similarly don't really add any new functionality, but they do give us a better way to group related pieces of information — we can do that with tuples, sort of, but tuples give us no way to indicate what their elements mean. (For example, a tuple of two `Int`s could represent a rational number or a point in 2D space.)

Slide 7

Case Classes

- Case classes are a very simple example of a *user-defined type* (analogous to predefined types such as `Int`, `String`, `List`, etc.).
- What they give you is a way to define a named type (e.g., `Rational`) that represents a collection of related objects (e.g., the numerator and denominator) and give the parts names:

```
case class Rational(numerator : Int, denominator : Int)
val r1 = Rational(1, 4)
println(r1.numerator + "/" + r1.denominator)
```

(Notice the dot? plays the same role, in a sense, as the one before, e.g., `toInt` or `map`.)

Slide 8

Case Classes, Continued

- You don't get the ability to define, within the class, operations for the type, but you could do that separately in functions:

```
def rationalToString(r : Rational) : String = {
  r.numerator + "/" + r.denominator
}
```

- (Is there a way to also define operations? yes, but for that you need a full-fledge class — "object orientation". POP II!)
- (Example(s)?)

Slide 9

Minute Essay

- Will you be in class next Tuesday?
- Are you planning to take POP II, either next semester or later?