Slide 1

Administrivia

- If you turned in code for Homework 2 that didn't work, I'd rather you fix it and resubmit. Your score will also be better than if I just grade the broken code.
- Homework 3 design due next Thursday, code the following Tuesday.
- Quiz solutions will be online shortly after class. I will also usually bring one printed solution you can look at after you turn in your paper.

Slide 2

Minute Essay From Last Lecture

- "Sample programs" has completed code for the two methods asked about.
- In the process of fixing one student's proposed solution, we discovered an even simpler way to do a case-insensitive sort of strings — `String` has a class variable (field) for the needed `Comparator`. !!

Slide 3

Quote of the Day/Week/?

- “As soon as we started programming, we found to our surprise that it wasn’t as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent finding mistakes in my own programs.” (Maurice Wilkes: 1948)

  (Wilkes was a key figure in the early days of computing.)

Slide 4

Homework 3

- In this homework you start writing code for your player, to replace the stick figure in the starter game.
- Key parts of this assignment are making the player
  - interact with different kinds of blocks.
  - move in response to keyboard or mouse input from human player.

  (If these don’t apply to your game, talk to me about whether there are reasonable substitutes.)

  For design phase, you just need to describe this interaction.

Slide 5

Homework 3, Continued

- `Player` defines some constants you should use.
- You will implement `KeyListener` or one/both of the mouse-listener interfaces. When you do this, the framework will deliver key and/or mouse “events” to you.
- Most logic will go in `update`, `getUpdateTime`, and the listener methods.

Slide 6

Error Handling — The Problem

- When you have a function in which something goes wrong, how do you tell the rest of the program?
- Examples:
  - Calling a square-root function with a negative number.
  - Trying to open (for reading) a file that doesn’t exist.
  - Trying to convert a string to an integer, when the string doesn’t contain something appropriate.

Slide 7

Error Handling — “Ostrich Approach”

- Idea — hope it doesn’t happen.
- Might sort of work if you tell users in your documentation, and maybe use assertions.
- But users make mistakes, and what then? e.g., out-of-bounds array access.
- And it may not always be easy to tell what inputs will produce errors (e.g., file access).

Slide 8

Error Handling — Return Codes

- Idea — have method return an error code if something goes wrong.
- Works well in situations where it might be hard to avoid sometimes causing the error.
- But requires that users of the method check for the “error” return value — tedious and error-prone.
- And what about methods that want to return a value? is it always possible to designate some value as “this means an error”?

Slide 9

Error Handling — Setting Flags

- Idea — have method set a flag somewhere if something goes wrong.
- Also useful in situations where it might be hard to avoid sometimes causing the error.
- Again, though, users have to check.
- Requires either an extra parameter (and changing it may be tricky in Java) or a “global” variable somewhere.

Slide 10

Error Handling — Exceptions

- Idea — when something goes wrong, “throw an exception”. What then?
- Aside — as program runs, we can think of it keeping a stack of nested method calls (“push” when we call a method, “pop” when one returns).
- When an exception is thrown, runtime system works its way up this stack until it finds something to “catch” the exception. If it never finds anything, it terminates the program (actually the thread).
- *Mostly* this is what Java library classes use to indicate errors — but some use return codes, so read documentation carefully.

Slide 11

Dealing With Exceptions

- Catching an exception — “try block”:

```
try { ....  }
catch (TypeOfException e) { ....  }
catch (OtherTypeOfException e) { ....  }
finally { .... } // optional
```

- Letting an exception “bubble up”:

```
void foo() throws WeirdException { .... }
```

- `Exception` class has some useful methods, e.g., `printStackTrace`.

Slide 12

Checked Versus Unchecked Exceptions

- “Checked exceptions” — ones that sensible programs are supposed to do something about (e.g., file not found).

  Must either catch these, or declare that your method lets them bubble up (and then callers must do likewise).

- “Unchecked exceptions” — ones for which maybe the reasonable thing to do is to just let the program crash.

  Can catch these, or let them bubble up (with or without declaration), possibly eventually crashing the program.

Slide 13

Throwing Exceptions

- Throwing an exception:

  `throw new TypeOfException(....)`
- Usually best to try to find an existing `Exception` class that fits, but can declare your own.
- Example — `withdraw` method in our bank account class. (Revisit this next time.)

Slide 14

Exceptions Versus Other Approaches

- What's the attraction?
  - Nice mechanism for dealing with errors and unexpected events.
  - Unlike return codes, can't just be ignored.
- But checked exceptions can be annoying to deal with . . .

Minute Essay

- None — quiz.