Administrivia

- Reminder: Quiz 3 next Tuesday.

- Reminder: Homework 3 design due next Tuesday.

- Code from Tuesday on Web, plus additional sorting/searching examples.

Slide 1

More About Arrays and Sorting

- In "instrumented sort" from last time, it might seem that `Sorter` and subclasses should be generic classes. This can be done, but is somewhat messy (several compile-time warnings) because of how generics are implemented under the hood.

Slide 2

- Writing your own sorting routines is pedagogically useful, but in practice you would probably use something from Java library.

- For arrays — `Arrays.sort` makes sense. Different versions ("polymorphism") allow sorting in "natural order" or in order you specify. Example: case-insensitive sorting of strings.

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 3

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 4

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 5

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 6

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.

Slide 7

- 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"?

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.

Slide 8

- Requires either an extra parameter (and changing it may be tricky in Java) or a "global" variable somewhere.

Error Handling — Exceptions

Slide 9

- 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.

Dealing With Exceptions

Slide 10

- 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`.

Checked Versus Unchecked Exceptions

Slide 11

- "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.

Throwing Exceptions

Slide 12

- 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.

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 . . .

Slide 13

Minute Essay

- Do you have questions about the code you wrote for Homework 2?

- Here's a line of code that can throw an exception:

```
String s;
double x;
/* code to get a value from the user and put in s omitted */
x = Double.parseDouble(s);
```

  Write a few lines of code to catch the kind of exception most likely to be

  thrown and print out a meaningful error message.

Slide 14

Slide 15

Minute Essay Answer

- First it would be useful to know kind of exception(s) could be thrown. You can
  find that out from the Java library API for class `Double`. `parseDouble`
  can throw a `NumberFormatException`, so you could write:

```
try {
    x = Double.parseDouble(s);
}
catch (NumberFormatException e) {
    System.out.println(s + " is not a number");
}
```