Slide 1

Administrivia

- Reminder: First quiz on Tuesday. About 10 minutes, at the end of class; open book/notes (including course Web site and Java API). 10 points, so not something to stress about.
- Reminder: Homework 1 code due Tuesday.
- Homework 2 due dates posted (design next Thursday, code following Tuesday).

Slide 2

Homework 1 Clarification(s)

- Far from unusual for students to feel a little lost at this point — so try on your own, then ask!
- Method `instance` in `BasicGameSetup` mentions "singleton". What's that about? Reference to "singleton design pattern" — idea that for some classes there should only ever be one instance.

Slide 3

Compiling and Running Java Programs, Revisited

- Recall discussion from a previous class about differences between Java and C with regard to compiling, linking, and executing — Java source code compiled to byte code, no linking step (such as is done for C), instead byte code for all classes loaded at runtime by the JVM.
- Where does the JVM find needed byte code? via "class path" — which can include
  - Directories/folders, such as the one(s) containing byte code for your classes — look for `.class` files.
  - "JAR (Java Archive)" files, which can contain byte code for many classes.

  Adding a JAR file to the default class path is a little like adding a flag such as `-lm` when you compile a C program.

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.
- 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 7

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 8

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 9

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 10

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 11

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.

Slide 12

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

Bank Account Example, Revisited

- Several methods need some sort of error handling — `withdraw` needs to do something if amount is too large, and both it and `deposit` could check for negative input. So let's fix that . . .
- And while we're making changes . . . (next slide). with `toString`, add "pay interest" method, improve test code, and do something slightly more interesting with the `HasPersonName` interface.

Slide 14

Bank Account Example, Other Changes

- Replace `print` with something more Java-idiomatic, `toString`.
- Improve test code (and put in a separate class).
- Revise class structure to allow more interesting use of `HasPersonName` interface — `Account` will have two subclasses, for personal accounts and for business accounts, with only the former implementing `HasPersonName`.
- Add method for paying interest. Goal here is to have all classes compute interest the same way, but allow different types of accounts to have different rates.

Slide 15

Minute Essay

- None — sign in.