

Slide 1

Administrivia

- Reminder: Homework 1 design due today (by midnight). Code due Tuesday. Updated information about office hours coming by e-mail.
- Quiz solutions will be on the Web shortly after class.

Slide 2

More Administrivia

- Two options for transferring files between machines:
 - Simple, not particularly smart: Just copy `.java` files between machines. Several options.
 - More complicated to get started, more professional: Use CVS (versioning software, Eclipse has built-in support).
- Some instructions in project description.

Compiling and Running Java Programs, Revisited

Slide 3

- 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.

Homework 1 Clarification(s)

Slide 4

- 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.

Error Handling — The Problem

Slide 5

- 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.

Error Handling — “Ostrich Approach”

Slide 6

- 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).

Error Handling — Return Codes

Slide 7

- 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”?

Error Handling — Setting Flags

Slide 8

- 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.

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

- None — quiz.

Slide 14