ure #12: Delegation, Exceptions, Assorted Features

es.

.

Trick: Delegation and Wrappers

ppropriate to use inheritance to extend something.

ives example of a TrReader, which *contains* another
vhich it *delegates* the task of actually going out and
acters.

mple: a class that instruments objects:

```
e {                    class Monitor implements Storage {
t x);                    int gets, puts;
                         private Storage store;
                         Monitor(Storage x) { store = x; gets = puts = 0; }
                         public void put(Object x) { puts += 1; store.put(x); }
                         public Object get() { gets += 1; return store.get(); }
                       }

                       // INSTRUMENTED
thing;                 Monitor S = new Monitor(something);
                       f(S);
                       System.out.println(S.gets + " gets");
```

led a *wrapper class.*

What to do About Errors?

t of any production program devoted to detecting and
o errors.

are external (bad input, network failures); others are
rs in programs.

d has stated precondition, it's the client's job to comply.

to detect and report client's errors.

throw exception objects, typically:

ew *SomeException* (*optional description*);

re objects. By convention, they are given two constructors:
arguments, and one with a descriptive string argument
xception stores).

throws some exceptions implicitly, as when you dereference
, or exceed an array bound.

Catching Exceptions

ses each active method call to *terminate abruptly,* until
we come to a **try** block.

tions and do something corrective with **try**:

f *that might throw exception;*
(*SomeException* e) {
omething reasonable;
(*SomeOtherException* e) {
omething else reasonable;

ith life;

Exception exception occurs during "Stuff..." and is not
e, we immediately "do something reasonable" and then
fe."

string (if any) available as e.getMessage() for error
d the like.

Catching Exceptions, II

rtype as the parameter type in a **catch** clause will catch
of that exception as well:

that might throw a FileNotFoundException or a
MalformedURLException ;
IOException ex) {
e any kind of IOException;

tFoundException and MalformedURLException both inherit
eption, the **catch** handles both cases.

eans that multiple **catch** clauses can apply; Java takes

it's nice to be more specific (concrete) about exception
possible.

, our style checker will therefore balk at the use of
RuntimeException, Error, and Throwable as exception

Catching Exceptions, III

atively new shorthand for handling multiple exceptions
y:

hat might throw IllegalArgumentException
IllegalStateException;
IllegalArgumentException|IllegalStateException ex) {
exception;

ceptions: Checked vs. Unchecked

hrown by **throw** command must be a subtype of Throwable g).

clares several such subtypes, among them

ed for serious, unrecoverable errors;

n, intended for all other exceptions;

xception, a subtype of Exception intended mostly for ing errors too common to be worth declaring.

l exceptions are all subtypes of one of these.

of Error or RuntimeException is said to be *unchecked.*

ception types are *checked.*

Unchecked Exceptions

r

er errors: many library functions throw rgumentException when one fails to meet a precondition.

tected by the basic Java system: e.g.,

ing x.y when x is null,

ing A[i] when i is out of bounds,

ing (String) x when x turns out not to point to a String.

tastrophic failures, such as running out of memory.

wn anywhere at any time with no special preparation.

Checked Exceptions

indicate exceptional circumstances that are not necessarily errors. Examples:

ng to open a file that does not exist.

utput errors on a file.

an interrupt.

ed exception that can occur inside a method must either y a try statement, or reported in the method's declaration.

```
 throws IOException, InterruptedException { ... }
```

nyRead (or something it calls) *might* throw IOException tedException.

sign: Why did Java make the following illegal?

```
{           class Child extends Parent {
 ... }          void f () throws IOException { ... }
            }
```


Good Practice

tions rather than using print statements and System.exit

esponse to a problem may depend on the *caller,* not just e problem arises.

v an exception when programmer violates preconditions.

good idea to throw an exception rather than let bad t a data structure.

document when methods throw exceptions.

formation about the cause of exceptional condition, put xception rather than into some global variable:

```
xtends Exception {            try {...
List errs;                    } catch (MyBad e) {
ist nums) { errs=nums; }       ... e.errs ...
                              }
```


Importing

.util.List every time you mean List or egex.Pattern every time you mean Pattern is annoying.

of the **import** clause at the beginning of a source file is previations:

ava.util.List; means "within this file, you can use List reviation for java.util.List.

ava.util.*; means "within this file, you can use *any* e in the package java.util without mentioning the package."

es *not* grant any special access; it *only* allows abbreviation.

ur program always contains import java.lang.*;

Static importing

lly get tired of writing System.out and Math.sqrt. Do eed to be reminded with each use that out is in the ystem package and that sqrt is in the Math package

es are of *static* members. A feature of Java allows you e such references:

tatic java.lang.System.out; means "within this file, e out as an abbreviation for System.out.

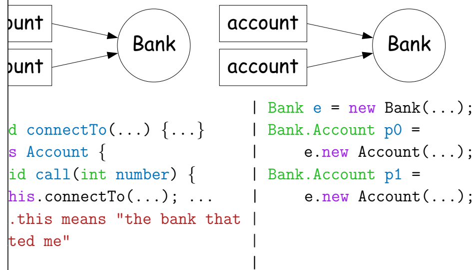
tatic java.lang.System.*; means "within this file, you y static member name in System without mentioning the

only an abbreviation. No special access.

't do this for classes in the anonymous package.

Inner Classes

- owed a static nested class. Static nested classes are other, except that they can be private or protected, see private variables of the enclosing class.
- ested classes are called *inner classes.*
- re (and syntax is odd); used when each instance of the is created by and naturally associated with an instance ining class, like Banks and Accounts:




```
                                 | Bank e = new Bank(...);
d connectTo(...) {...}           | Bank.Account p0 =
s Account {                      |     e.new Account(...);
id call(int number) {            | Bank.Account p1 =
his.connectTo(...); ...          |     e.new Account(...);
.this means "the bank that       |
ted me"                          |
                                 |
```


Nesting Classes

- t makes sense to *nest* one class in another. The nested
  - nly in the implementation of the other, or
  - tually "subservient" to the other
- classes can help avoid name clashes or "pollution of the with names that will never be used anywhere else.
- lynomials can be thought of as sequences of terms. meaningful outside of Polynomials, so you might define present a term *inside* the Polynomial class:

```
ial {

olynomials

m[] terms;
tic class Term {
```


Type testing: instanceof

- to ask about the dynamic type of something:

```
ker(Reader r) {
nceof TrReader)
t.print("Translated characters: ");

t.print("Characters: ");
```

- s is seldom what you want to do. Why do this:

```
eof StringReader)
StringReader) x;
stanceof FileReader)
FileReader) x;
```

- just call `x.read()`?!
- se instance methods rather than **instanceof**.