Lecture #12: Additional OOP Details, Exceptions

To Think About

lds a JUnit test:

```
oid mogrifyTest() {
tEquals("mogrify fails",
        new int[] { 2, 4, 8, 12 },
        MyClass.mogrify(new int[] { 1, 2, 4, 6 }));
```

ays seems to fail, no matter what `mogrify` does. Why?

es this in an autograder log:

roj0/signpost directory.

y to be the problem?

es not see his proj0 submission under the Scores tab.
the problem?

Parent Constructors

tes #5, talked about how Java allows implementer of a
rol all manipulation of objects of that class.

, this means that Java gives the constructor of a class
t at each new object.

ass extends another, there are two constructors—one
nt type and one for the new (child) type.

Java guarantees that one of the parent's constructors
t. In effect, there is a call to a parent constructor at
g of every one of the child's constructors.

he parent's constructor yourself. By default, Java calls
" (parameterless) constructor.

```
re {                    class Rectangle extends Figure {
igure(int sides) {         public Rectangle() {
                               super(4);
                           }...
                        }
```


Using an Overridden Method

t you wish to *add* to the action defined by a superclass's
ıer than to completely override it.

ng method can refer to overridden methods by using
refix super.

, you have a class with expensive functions, and you'd
zing version of the class.

```
uteHard {
cate(String x, int y) { ... }


uteLazily extends ComputeHard {
cate(String x, int y) {
n't already have answer for this x and y) {
 result = super.cogitate(x, y);  // <<< Calls overridden function
noize (save) result;
urn result;

 memoized result;
```


Trick: Delegation and Wrappers

ppropriate to use inheritance to extend something.

ives example of a `TrReader`, which *contains* another
which 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:

SomeException (optional description);

re objects. By convention, they are given two construc-
h no arguments, and one with a descriptive string argu-
the exception stores).

throws some exceptions implicitly, as when you deref-
pointer, or exceed an array bound.

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 ;
Exception ex) {
e any kind of IOException;
```

tFoundException and MalformedURLException both in-OException, the **catch** handles both cases.

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

it's nice to be more (concrete) about exception types le.

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

Catching Exceptions

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

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

```
at might throw exception;
meException e) {
hing reasonable;
meOtherException e) {
hing else reasonable;

fe;
```

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

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

ceptions: Checked vs. Unchecked

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

lares 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.*

Catching Exceptions, III

latively new shorthand for handling multiple exceptions y:

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


Checked Exceptions

indicate exceptional circumstances that are not neces-mmer errors. Examples:

g to open a file that does not exist.

utput errors on a file.

an interrupt.

ed exception that can occur inside a method must ei-dled by a try statement, or reported in the method's

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

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

sign: Why did Java make the following illegal?

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


Unchecked Exceptions

er errors: many library functions throw rgumentException when one fails to meet a precondi-

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.

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.

w 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 {
List errs;
ist nums) { errs=nums; }
```

```
try {...
} catch (MyBad e) {
 ... e.errs ...
}
```