CS61B Lecture #12: 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/galaxy directory.`

y to be the problem?

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

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

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

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

```
that might throw exception;
(SomeException e) {
nething reasonable;
(SomeOtherException e) {
nething else reasonable;

life;
```

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.

Catching Exceptions, II

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

```
de that might throw a FileNotFoundException or a
  MalformedURLException ;
(IOException ex) {
ndle 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, III

latively new shorthand for handling multiple exceptions
y:

```
e that might throw IllegalArgumentException
or IllegalStateException;
(IllegalArgumentException|IllegalStateException ex) {
dle exception;
```

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.

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.*

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:

```
l extends Exception {
IntList errs;
ntList nums) { errs=nums; }
```

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


Checked Exceptions

indicate exceptional circumstances that are not neces-
mmer 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 ei-
dled by a `try` statement, or reported in the method's

```
i() throws IOException, InterruptedException { ... }
```

yRead (or something it calls) *might* throw `IOException`
tedException.
sign: Why did Java make the following illegal?

```
nt {
) { ... }
```

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