To Think About

• A student adds a JUnit test:

```java
@Test
public void mogrifyTest() {
    assertEquals("mogrify fails", new int[] { 2, 4, 8, 12 },
        MyClass.mogrify(new int[] { 1, 2, 4, 6 }));
}
```

The test always seems to fail, no matter what `mogrify` does. Why?

• A student sees this in an autograder log:

```
Fatal: no proj0/galaxy directory.
```

What is likely to be the problem?

• A student does not see his proj0 submission under the Scores tab. What can be the problem?
CS61B Lecture #12: Exceptions
What to do About Errors?

• Large amount of any production program devoted to detecting and responding to errors.

• Some errors are external (bad input, network failures); others are internal errors in programs.

• When method has stated precondition, it’s the client’s job to comply.

• Still, it’s nice to detect and report client’s errors.

• In Java, we throw exception objects, typically:

  throw new SomeException (optional description);

• Exceptions are objects. By convention, they are given two constructors: one with no arguments, and one with a descriptive string argument (which the exception stores).

• Java system throws some exceptions implicitly, as when you dereference a null pointer, or exceed an array bound.
Catching Exceptions

• A **throw** causes each active method call to *terminate abruptly*, until (and unless) we come to a **try** block.

• Catch exceptions and do something corrective with **try**:

  ```java
try {
      Stuff that might throw exception;
  }
catch (SomeException e) {
      Do something reasonable;
  }
catch (SomeOtherException e) {
      Do something else reasonable;
  }
  Go on with life;
  ```

• When **SomeException** exception occurs during “Stuff…” and is not handled there, we immediately “do something reasonable” and then “go on with life.”

• Descriptive string (if any) available as e.getMessage() for error messages and the like.
Catching Exceptions, II

- Using a supertype as the parameter type in a `catch` clause will catch any subtype of that exception as well:

```java
try {
    Code that might throw a FileNotFoundException or a MalformedURLException;
    catch (IOException ex) {
        Handle any kind of IOException;
    }
}
```

- Since `FileNotFoundException` and `MalformedURLException` both inherit from `IOException`, the `catch` handles both cases.

- Subtyping means that multiple `catch` clauses can apply; Java takes the first.

- Stylistically, it’s nice to be more (concrete) about exception types where possible.

- In particular, our style checker will therefore balk at the use of `Exception`, `RuntimeException`, `Error`, and `Throwable` as exception supertypes.
Catching Exceptions, III

• There’s a relatively new shorthand for handling multiple exceptions the same way:

```java
try {
    Code that might throw IllegalArgumentException
    or IllegalStateException;
} catch (IllegalArgumentException | IllegalStateException ex) {
    Handle exception;
}
```
Exceptions: Checked vs. Unchecked

• The object thrown by `throw` command must be a subtype of `Throwable` (in `java.lang`).

• Java pre-declares several such subtypes, among them
  - `Error`, used for serious, unrecoverable errors;
  - `Exception`, intended for all other exceptions;
  - `RuntimeException`, a subtype of `Exception` intended mostly for programming errors too common to be worth declaring.

• Pre-declared exceptions are all subtypes of one of these.

• Any subtype of `Error` or `RuntimeException` is said to be `unchecked`.

• All other exception types are `checked`. 
Unchecked Exceptions

• Intended for
  
  - Programmer errors: many library functions throw 
    IllegalArgumentException when one fails to meet a precondi-
    tion.
  
  - Errors detected by the basic Java system: e.g.,
    * Executing x.y when x is null,
    * Executing A[i] when i is out of bounds,
    * Executing (String) x when x turns out not to point to a String.

  - Certain catastrophic failures, such as running out of memory.

• May be thrown anywhere at any time with no special preparation.
Checked Exceptions

• Intended to indicate exceptional circumstances that are not necessarily programmer errors. Examples:
  - Attempting to open a file that does not exist.
  - Input or output errors on a file.
  - Receiving an interrupt.
• Every checked exception that can occur inside a method must either be handled by a `try` statement, or reported in the method's declaration.
• For example,

  ```java
  void myRead() throws IOException, InterruptedException {
  ...
  }
  ```

  means that myRead (or something it calls) *might* throw IOException or InterruptedException.

• Language Design: Why did Java make the following illegal?

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

• Throw exceptions rather than using print statements and System.exit everywhere,

• ... because response to a problem may depend on the caller, not just method where problem arises.

• Nice to throw an exception when programmer violates preconditions.

• Particularly good idea to throw an exception rather than let bad input corrupt a data structure.

• Good idea to document when methods throw exceptions.

• To convey information about the cause of exceptional condition, put it into the exception rather than into some global variable:

```java
class MyBad extends Exception {
    public IntList errs;
    MyBad(IntList nums) { errs=nums; }
    try {
        ... e.errs ...
    }
```