To Think About

- A student adds a JUnit test:

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

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

- A student sees this in an autograder log:

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

  What is likely to be the problem?

- A student does not see his proj0 submission under the Sco
  What can be the problem?

CS61B Lecture #12: Additional OOP Details Exceptions

Parent Constructors

- In lecture notes #5, talked about how Java allows implemen class to control all manipulation of objects of that class.
- In particular, this means that Java gives the constructor of the first shot at each new object.
- When one class extends another, there are two constructo for the parent type and one for the new (child) type.
- In this case, Java guarantees that one of the parent's const is called first. In effect, there is a call to a parent constru the beginning of every one of the child's constructors.
- You can call the parent's constructor yourself. By default, Ja the "default" (parameterless) constructor.

```
class Figure {
  public Figure(int sides) {
     ...
  }...
}
```

```
class Rectangle extends
   public Rectangle() {
      super(4);
   }...
}
```

Using an Overridden Method

- Suppose that you wish to *add* to the action defined by a supe method, rather than to completely override it.
- The overriding method can refer to overridden methods b the special prefix super.
- For example, you have a class with expensive functions, a like a memoizing version of the class.

```
class ComputeHard {
  int cogitate(String x, int y) { ... }
}

class ComputeLazily extends ComputeHard {
  int cogitate(String x, int y) {
    if (don't already have answer for this x and y) {
       int result = super.cogitate(x, y);  // <<< Calls ove
       memoize (save) result;
       return result;
    }
     return memoized result;
  }
}
```

Trick: Delegation and Wrappers

- Not always appropriate to use inheritance to extend someth
- Homework gives example of a `TrReader`, which *contains* `Reader`, to which it *delegates* the task of actually going reading characters.
- Another example: a class that instruments objects:

```
interface Storage {
  void put(Object x);
  Object get();
}
```

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

```
// ORIGINAL
Storage S = something;
f(S);
```

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

`Monitor` is called a *wrapper class.*

What to do About Errors?

- Large amount of any production program devoted to detec
  responding to errors.
- Some errors are external (bad input, network failures); oth
  internal errors in programs.
- When method has stated precondition, it's the client's job to
- 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 co
  tors: one with no arguments, and one with a descriptive stri
  ment (which the exception stores).
- Java system throws some exceptions implicitly, as when you
  erence a null pointer, or exceed an array bound.

Catching Exceptions

- A **throw** causes each active method call to *terminate abrupt*
  (and unless) we come to a **try** block.
- Catch exceptions and do something corrective with **try**:

```
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..." ar
  handled there, we immediately "do something reasonable" a
  "go on with life."
- Descriptive string (if any) available as `e.getMessage()` fo
  messages and the like.

Catching Exceptions, II

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

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

- Since `FileNotFoundException` and `MalformedURLException`
  herit from *IOException*, the **catch** handles both cases.
- Subtyping means that multiple **catch** clauses can apply; Jav
  the first.
- Stylistically, it's nice to be more (concrete) about exceptio
  where possible.
- In particular, our style checker will therefore balk at the
  `Exception`, `RuntimeException`, `Error`, and `Throwable` as ex
  supertypes.

Catching Exceptions, III

- There's a relatively new shorthand for handling multiple exc
  the same way:

```
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 `T`
  (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 mo
    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 *unc*
- All other exception types are *checked.*

Unchecked Exceptions

- Intended for
  - Programmer errors: many library functions throw `IllegalArgumentException` when one fails to meet a pr 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
  - Certain catastrophic failures, such as running out of mem
- May be thrown anywhere at any time with no special prepara

Checked Exceptions

- Intended to indicate exceptional circumstances that are not sarily 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 m ther be handled by a `try` statement, or reported in the m declaration.
- For example,

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

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

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

```
class Parent {
   void f() { ... }
}
```

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

Good Practice

- Throw exceptions rather than using print statements and Sy everywhere,
- . . . because response to a problem may depend on the *caller,* method where problem arises.
- Nice to throw an exception when programmer violates precor
- Particularly good idea to throw an exception rather than input corrupt a data structure.
- Good idea to document when methods throw exceptions.
- To convey information about the cause of exceptional condit it into the exception rather than into some global variable:

```
class MyBad extends Exception {
   public IntList errs;
   MyBad(IntList nums) { errs=nums; }
}
```

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