FINAL REVIEW

**Stronger vs Weaker**

- Requires more?  
  weaker
- Promises more? (stricter specifications on what the effects entail)  
  stronger

**Subtypes & Subclasses**

- Subtypes are substitutable for supertypes  
  if Foo is a subtype of Bar, G<Foo> is a NOT a subtype of G<Bar>  
  - Aliasing resulting from this would let you add objects of type Bar to G<Foo>, which would be bad!  
  - Example:  
    List<String> ls = new ArrayList<String>();  
    List<Object> lo = ls;  
    lo.add(new Object());  
    String s = ls.get(0);  
- Subclassing is done to reuse code (extends)  
  - A subclass can override methods in its superclass

**Typing and Generics**

- <$> is a wildcard for unknown  
  - Upper bounded wildcard: type is wildcard or subclass  
    - Eg Lists? extends Shape  
    - Illegal to write into (no calls to add!) because we can’t guarantee type safety  
  - Lower bounded wildcard: type is wildcard or superclass  
    - Eg Lists? super Integer  
    - May be safe to write into.

**Subtypes & Subclasses**

- class Student extends Object {...}  
  class CSEStudent extends Student {...}  

   List<Student> ls;  
   List<?> extends Student> les;  
   List<?> super Student> lss;  
   List<CSEStudent> lcse;  
   List<?> extends CSEStudent> lcse;  
   List<?> super CSEStudent> lcsce;  
   Student scholar;  
   CSEStudent hacker;
Subclasses & Overriding

class Foo extends Object {
    Shoe m(Shoe x, Shoe y){ ... }
}

class Bar extends Foo {...}

Method Declarations in Bar

<table>
<thead>
<tr>
<th>Method Declaration</th>
<th>Overriding</th>
<th>Overloading</th>
<th>Type-Error</th>
<th>None (new method declaration)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar::m(Shoe x, Shoe y) {...}</td>
<td>overriding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bar::m(Shoe q, Shoe z) {...}</td>
<td>overriding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bar::m(FootWear x, Shoe y) {...}</td>
<td>overloading</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bar::m(FootWear x, HighHeeledShoe y) {...}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bar::m(FootWear x, FootWear y) {...}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bar::m(Shoe x, Shoe y) {...}</td>
<td>overloading</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bar::m(Shoe x, Shoe y) {...}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bar::m(FootWear x, HighHeeledShoe y) {...}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bar::m(Shoe y) {...}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bar::z(Shoe x, Shoe y) {...}</td>
<td>none (new method declaration)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Design Patterns

- Creational patterns: get around Java constructor inflexibility
  - Sharing: singleton, interning, flyweight
  - Returning a subtype: factories
- Structural patterns: translate between interfaces
  - Adapter: same functionality, different interface
  - Decorator: different functionality, same interface
  - Proxy: same functionality, same interface, restrict access
  - All of these are types of wrappers

Design Patterns

- Interpreter pattern:
  - Collects code for similar objects, spreads apart code for operations
    - Classes for objects with operations as methods in each class
  - Instance of Composite pattern
- Procedural patterns:
  - Collects code for similar operations, spreads apart code for objects
    - Classes for operations, method for each operand type
  - Easy to add methods, hard to add objects
  - Ex: Visitor pattern

Design Patterns

- What pattern would you use to...
  - add a scroll bar to an existing window object in Swing
    - Decorator
  - We have an existing object that controls a communications channel. We would like to provide the same interface to clients but transmit and receive encrypted data over the existing channel.
    - Proxy
  - When the user clicks the "find path" button in the Campus Maps application (hw9), the path appears on the screen.
    - MVC
    - Observer