**Final Review**

---

### Stronger vs Weaker (one more time!)

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

### Stronger vs Weaker

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Stronger</th>
<th>Weaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>@requires snurf flarg</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>@modifies blech murph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@effects roar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@throws woiefio fi gonzoe</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **A.** @modifies blech murph @effects roar @throws woiefio fi gonzoe

- **B.** @requires snurf flarg @modifies blech murph @throws woiefio fi gonzoe

- **C.** @requires snurf flarg @modifies blech murph @effects roar

### 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:**
    ```java
    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: List<? 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: List<? super Integer>`
    - May be safe to write into.

### Subtypes & Subclasses

```java
class Student extends Object { ... }
class CSEStudent extends Student { ... }
```

```java
List<Student> ls;
List<? extends Student> les;
List<? super Student> lss;
List<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```
Subclasses & Overriding

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

class Bar extends Foo {...}

Method Declarations in Bar

- The result is method overriding
- The result is method overloading
- The result is a type-error
- None of the above

<table>
<thead>
<tr>
<th></th>
<th>Object</th>
<th>Footwear</th>
<th>Shoe</th>
<th>HighHeeledShoe</th>
</tr>
</thead>
<tbody>
<tr>
<td>FootWear m(Shoe x, Shoe y) { ... }</td>
<td>type-error</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shoe m(Shoe q, Shoe z) { ... }</td>
<td>overriding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HighHeeledShoe m(Shoe x, Shoe y) { ... }</td>
<td>overriding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shoe m(FootWear x, HighHeeledShoe y) { ... }</td>
<td>overloading</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shoe m(FootWear x, FootWear y) { ... }</td>
<td>overloading</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shoe m(Shoe x, Shoe y) { ... }</td>
<td>overloading</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shoe m(HighHeeledShoe x, HighHeeledShoe y) { ... }</td>
<td>overloading</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shoe m(Shoe y) { ... }</td>
<td>overloading</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shoe 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
  - Telescoping constructor fix: builder
  - 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

- Interpreter pattern:
  - Collects code for similar objects, spreads apart code for operations (classes for objects with operations as methods in each class)
  - Easy to add objects, hard to add methods
  - 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

Adapter, Builder, Composite, Decorator, Factory, Flyweight, Iterator, Intern, Interpreter, Model-View-Controller (MVC), Observer, Procedural, Prototype, Proxy, Singleton, Visitor, Wrapper