FINAL REVIEW
Stronger vs Weaker (one more time!)

- Requires more?
  
  weaker

- Promises more? (strict specifications on what the effects entail)
  
  stronger
Stronger vs Weaker

A. @requires key is a key in this and key != null
   @return the value associated with key
   @throws NullPointerException if key is null
   WEAKER

B. @return the value associated with key if key is a key in this, or null if key is not associated with any value
   NEITHER

C. @return the value associated with key
   @throws NullPointerException if key is null
   @throws NoSuchElementException if key is not a key this
   STRONGER
Subtypes & Subclasses

- Subtypes are substitutable for supertypes
- If \texttt{Foo} is a subtype of \texttt{Bar}, \texttt{G<Foo>} is a \textbf{NOT} a subtype of \texttt{G<Bar>}
  - Aliasing resulting from this would let you add objects of type \texttt{Bar} to \texttt{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

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> lecse;
List<? super CSEStudent> lscse;

Student scholar;
CSEStudent hacker;

ls = lcse;  \[\text{x}\]
les = lscse; \[\text{x}\]
lcse = lscse; \[\text{x}\]
lecse.add(scholar); \[\text{x}\]
lscse.add(scholar); \[\text{x}\]
lss.add(hacker); \[\text{✓}\]
scholar = lscse.get(0); \[\text{x}\]
hacker = lecse.get(0); \[\text{✓}\]
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>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>FootWear (m(\text{Shoe} \ x, \ \text{Shoe} \ y)) { ... }</td>
<td>type-error</td>
</tr>
<tr>
<td>Shoe (m(\text{Shoe} \ q, \ \text{Shoe} \ z)) { ... }</td>
<td>overriding</td>
</tr>
<tr>
<td>HighHeeledShoe (m(\text{Shoe} \ x, \ \text{Shoe} \ y)) { ... }</td>
<td>overriding</td>
</tr>
<tr>
<td>Shoe (m(\text{FootWear} \ x, \ \text{HighHeeledShoe} \ y)) { ... }</td>
<td>overloading</td>
</tr>
<tr>
<td>Shoe (m(\text{FootWear} \ x, \ \text{FootWear} \ y)) { ... }</td>
<td>overloading</td>
</tr>
<tr>
<td>Shoe (m(\text{Shoe} \ x, \ \text{Shoe} \ y)) { ... }</td>
<td>overriding</td>
</tr>
<tr>
<td>Shoe (m(\text{HighHeeledShoe} \ x, \ \text{HighHeeledShoe} \ y)) { ... }</td>
<td>overloading</td>
</tr>
<tr>
<td>Shoe (m(\text{Shoe} \ y)) { ... }</td>
<td>overloading</td>
</tr>
<tr>
<td>Shoe (z(\text{Shoe} \ x, \ \text{Shoe} \ y)) { ... }</td>
<td>none (new method declaration)</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
Design Patterns

• 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