



FINAL REVIEW

Stronger vs Weaker (one more time!)

- Requires more?

weaker

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

stronger

Stronger vs Weaker

```
@requires key is a key in this  
@return the value associated with key  
@throws NullPointerException if key is null
```

- A. @requires key is a key in this and key != null
@return the value associated with key **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 `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: `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; X  
les = lscse; X  
lcse = lscse; X  
les.add(scholar); X  
lscse.add(scholar); X  
lss.add(hacker);   
scholar = lscse.get(0); X  
hacker = lecse.get(0); 
```

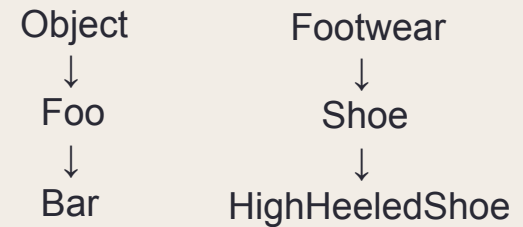
Subclasses & Overriding

```
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



- FootWear m(Shoe x, Shoe y) { ... } **type-error**
- Shoe m(Shoe q, Shoe z) { ... } **overriding**
- HighHeeledShoe m(Shoe x, Shoe y) { ... } **overriding**
- Shoe m(FootWear x, HighHeeledShoe y) { ... } **overloading**
- Shoe m(FootWear x, FootWear y) { ... } **overloading**
- Shoe m(Shoe x, Shoe y) { ... } **overriding**
- Shoe m(HighHeeledShoe x, HighHeeledShoe y) { ... } **overloading**
- Shoe m(Shoe y) { ... } **overloading**
- Shoe z(Shoe x, Shoe y) { ... } **none (new method declaration)**

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

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

- 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