
CSE 331

Software Design & Implementation

Winter 2022

Section 10: Review + Design Patterns

Administrivia

- HW9 due ~~today~~ Saturday 3/12 @ 11:00pm – everyone has 2 free late days to finish up!
 - Please volunteer to show off your projects!
- Final on Tuesday 12:30 pm – same room assignments as midterm
 - Review Session Monday 4:30 pm, CSE2 G20 auditorium
- Any questions?

Agenda

- Review
Reasoning, Specifications, ADTs (RI & AF), Testing, Defensive Programming, Equals and Hash Code, Exceptions, Subtyping, Generics
- Design Patterns

Stronger vs Weaker (one more time!)

- In each case, what is the effect of changing the amount of information required about the input?
- Requires more about inputs?
- Promises more about behavior?

Stronger vs Weaker (one more time!)

- In each case, what is the effect of changing the amount of information required about the input?
- Requires more about inputs?

weaker

- Promises more about behavior?

stronger

Stronger vs Weaker

Compared to the spec in the box, what is the effect of using specs A,B,C in terms of our statement's strength (weaker/stronger/neither)?

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

- A. @requires that key is a key in this and key != null
@return the value associated with key
- B. @return the value associated with key if key is a key in *this*, or null if key is not associated with any value
- C. @return the value associated with key
@throws NullPointerException if key is null
@throws NoSuchElementException if key is not a key *this*

Stronger vs Weaker

Compared to the spec in the box, what is the effect of using specs A,B,C in terms of our statement's strength (weaker/stronger/neither)?

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

A. @requires that 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**

Exceptions

- Unchecked exceptions are ignored by the compiler.
- If a method throws a checked exception or calls a method that throws a checked exception, then it must either:
 - catch the exception
 - declare it in `@throws`

Exceptions Examples

Should these be checked or unchecked?

- Attempt to write an invalid type into an array
E.g., write `Double` into `Integer[]` cast to `Number[]`
- Attempt to open a file that does not exist
- Attempt to create a URL from invalidly formatted text
E.g., “http:/foo” (only one “/”)

Exceptions Examples

Should these be checked or unchecked?

- Attempt to write an invalid type into an array
E.g., write `Double` into `Integer[]` cast to `Number[]`
unchecked
- Attempt to open a file that does not exist
checked
- Attempt to create a URL from invalidly formatted text
E.g., “http:/foo” (only one “/”)
debatable – could see either one

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>`
 - Safe to read from: result will be a `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.
 - Illegal to retrieve as type other than `Object`.

Subtypes & Subclasses

Given the below classes which one of the statements in the box are legal?

```
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;
```

```
les = lscse;
```

```
lcse = lscse;
```

```
les.add(scholar);
```

```
lscse.add(scholar);
```

```
lss.add(hacker);
```

```
scholar = lscse.get(0);
```

```
hacker = lecse.get(0);
```

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;  
lcse = lscse;  
les.add(scholar);  
lscse.add(scholar);  
lss.add(hacker);  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```

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 = lcse;  
les.add(scholar);  
lscse.add(scholar);  
lss.add(hacker);  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```

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);  
lscse.add(scholar);  
lss.add(hacker);  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```


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);  
lss.add(hacker);  
scholar = lscse.get(0);  
hacker = lecse.get(0);
```

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);  
hacker = lecse.get(0);
```

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);  
hacker = lecse.get(0);
```

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);
```

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); 😊
```

equals for a parameterized class

```
class Node<E> {
```

```
...
```

```
@Override
```

```
public boolean equals(Object obj) {
```

```
    if (!(obj instanceof Node<?>)) {
```

```
        return false;
```

```
    }
```

```
    Node<?> n = (Node<?>) obj;
```

```
    return this.data().equals(n.data());
```

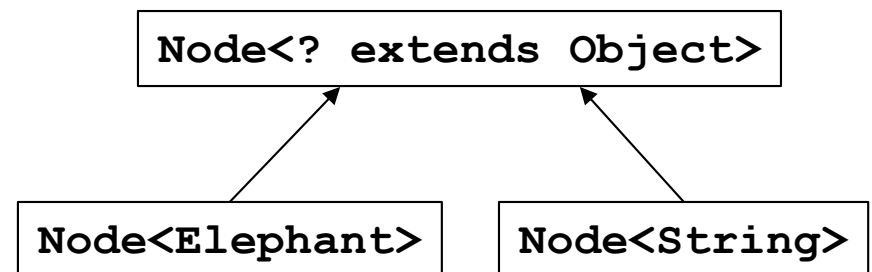
```
}
```

```
...
```

```
}
```

Works if the type of obj is Node<Elephant> or Node<String> or ...

Leave it to here to “do the right thing” if `this` and `n` differ on element type



Subclasses & Overriding

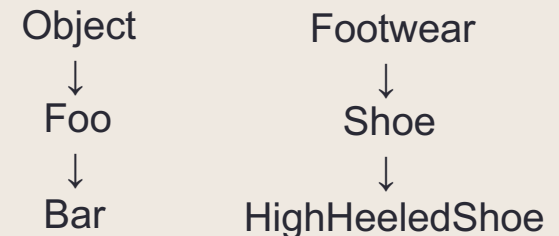
```
class Foo extends Object {  
    Shoe m(Shoe x, Shoe y) { ... }  
}
```

```
class Bar extends Foo {...}
```

Method Declarations in Bar

Given the class in the purple box, determine whether the method declarations for the method Shoe() inside Bar class are overriding or overloading it?

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

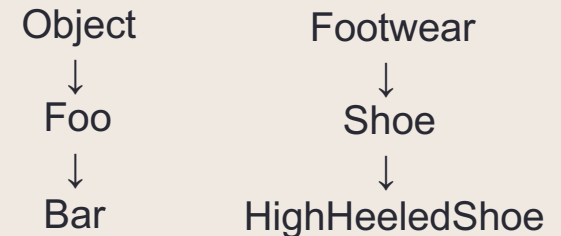


- FootWear m(Shoe x, Shoe y) { ... }
- Shoe m(Shoe q, Shoe z) { ... }
- HighHeeledShoe m(Shoe x, Shoe y) { ... }
- Shoe m(FootWear x, HighHeeledShoe y) { ... }
- Shoe m(FootWear x, FootWear y) { ... }
- Shoe m(Shoe x, Shoe y) { ... }
- Shoe m(HighHeeledShoe x, HighHeeledShoe y) { ... }
- Shoe m(Shoe y) { ... }
- Shoe z(Shoe x, Shoe y) { ... }

```
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)**

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}
class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}
class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

Given the declarations on the left, determine the outcome of the expressions below.

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```

```
Bird b = new Duck();
b.move(42);
```

```
Bird b = new RubberDuck();
b.move(3);
```

```
Duck donald = new RubberDuck();
donald.swim();
```

```
Duck donald = new RubberDuck();
donald.move();
```

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}
class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}
class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```

Compile error: cannot create instances of an abstract class.

```
b.move(42);
```

```
Bird b = new RubberDuck();
b.move(3);
```

```
Duck donald = new RubberDuck();
donald.swim();
```

```
Duck donald = new RubberDuck();
donald.move();
```

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}
class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}
class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```



Chirp!
Chirp!

```
Bird b = new RubberDuck();
b.move(3);
```

```
Duck donald = new RubberDuck();
donald.swim();
```

```
Duck donald = new RubberDuck();
donald.move();
```

Subclasses & Method Overriding

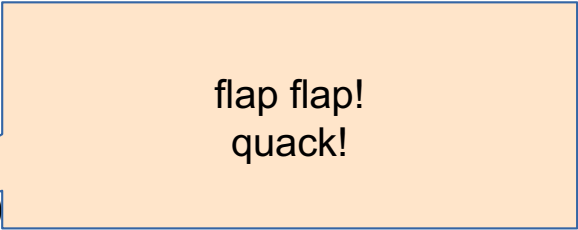
```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}
class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}
class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("plop!"); }
}
```

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```

```
Bird b = new RubberDuck();
b.move(42);
```

```
Bird b = new RubberDuck();
b.move(3);
```



flap flap!
quack!

```
donald.swim();
```

```
Duck donald = new RubberDuck();
donald.move();
```

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}
class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}
class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```

```
Bird b = new Duck();
b.move(42);
```

```
Bird b = new RubberDuck();
b.move(2);
```

squeak!
paddle!
squeak!

donald.move();

RubberDuck();

RubberDuck();

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}
class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}
class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); }
    public void swim() { System.out.println("swim!"); }
}
```

Compile error: no swim method
in class Duck

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```

```
Bird b = new Duck(),
b.move(42);
```

```
Bird b = new RubberDuck();
b.move(3);
```

```
Duck donald = new RubberDuck();
donald.swim();
```

```
Duck donald = new RubberDuck();
donald.move();
```

Subclasses & Method Overriding

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}
class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}
class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}
class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

```
Bird b = new Bird();
b.move();
```

```
Bird b = new Canary();
b.move(17);
```

```
Bird b = new
b.move(42);
```

```
Bird b = new RubberDuck();
b.move(3);
```

squeak!
paddle!

```
Donald = new RubberDuck();
donald.move();
```

```
Donald = new RubberDuck();
donald.move();
```


Event-Driven Programs

- Sits in an event loop, waiting for events to process
 - often does so until forcibly terminated
- Two common types of event-driven programs:
 - GUIs
 - Web servers
- Where is the event loop in Spark Java?
 - it is created behind the scenes

Design Patterns

Design Patterns

- Creational patterns: get around Java constructor inflexibility
 - Sharing: singleton, interning
 - 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, 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
 - 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.
 - When the user clicks the “find path” button in the Campus Maps application (hw9), the path appears on the screen.

Design Patterns

Adapter, Builder, Composite, Decorator, Factory, 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**