CSE 331
Software Design & Implementation

Winter 2023

Section 10: Review

Administrivia

- HW9 due ~~today~~ Saturday 3/11 @ 11 PM – everyone has 2 free late days to finish up!
  - Please volunteer to show off your projects in lecture tomorrow!
- Final on Tuesday 3/14 at 12:30 – same rooms as midterm: GWN 201 (A-I), GWN 301 (J-Z)
  - Review session on Monday (3/13) at 4:30 PM in JHN 102. Come with questions!
- 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

ADT Specifications

- Any ADT should have a complete specification of its behavior, including:
  - An abstract description of the class
  - A specification for each method, detailing the behavior of a method in **all** cases

- We can use JavaDoc to write these specifications
  - There are strict rules: there needs to be a top-level description of the method, every parameter needs an @param, etc.

- Do not refer to the implementation (i.e. fields) of the class in these specifications!

ADT Implementation

- To implement the ADT, we need a set of fields to store its state.
- Somehow, these fields represent the abstract definition of the ADT. This translation from the fields to the abstract data is called the **abstraction function**.
- In order for an instance to be "valid," there are certain restrictions on the fields. We call this the **representation invariant**
  - e.g. certain fields might not be able to be null, or size has to be greater than or equal to 0
  - We use a private method `checkRep` to verify this.
- Both your AF and RI need to mention your fields!

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` in the method signature

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

Testing

What would be some good test cases for this method?

Note: @param tags omitted.

```
/** A very mysterious method with a great description
 *  @requires x > 0 and y > 0
 *  @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
  if (x == y) { throw new IllegalArgumentException(); }
  return (x > y) ? (x + y) : (x - y);
}
```

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 *  @requires x > 0 and y > 0
 *  @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
  if (x == y) { throw new IllegalArgumentException(); }
  return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(0, 0)` a good test case?

No. Its behavior is undefined. We cannot test for undefined behavior!

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 *  @requires x > 0 and y > 0
 *  @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x – y if x < y
public int mystery(int x, int y) {
  if (x == y) { throw new IllegalArgumentException(); }
  return (x > y) ? (x + y) : (x – y);
}
```

Is `mystery(1, 1)` a good test case?

Yes – we are testing for an IllegalArgumentException being thrown.

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 *  @requires x > 0 and y > 0
 *  @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
  if (x == y) { throw new IllegalArgumentException(); }
  return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(3, 2)` a good test case?

Yes!

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 *  @requires x > 0 and y > 0
 *  @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
  if (x == y) { throw new IllegalArgumentException(); }
  return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(4, 10)` a good test case?

Yes!

Testing

What would be some good test cases for this method?

```
/** A very mysterious method with a great description
 *  @requires x > 0 and y > 0
 *  @throws IllegalArgumentException if x == y
 */ @return x + y if x > y. x - y if x < y
public int mystery(int x, int y) {
  if (x == y) { throw new IllegalArgumentException(); }
  return (x > y) ? (x + y) : (x - y);
}
```

Is `mystery(42, -42)` a good test case?

No! It’s testing for undefined behavior!

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
  - Lower bounded wildcard **`<? super SomeClass>`** (superclass)
    - Can only insert items with type **`SomeClass`**, or a type that extends **`SomeClass`**
      - Why? Because we can cast that object into **`SomeClass`**.
    - Illegal to retrieve as type other than **`Object`**.

- What types can you put here?
  - **List<? super Number> lsn = new ArrayList<________>();**
    - Object
    - Number
  - Number *is* an Object. But an Object might not be a Number.

Typing and Generics

- What types can you put here?
  - **List<? super Number> lsn = new ArrayList<_________>();**
    - Object
    - Number
  - Number *is* an Object. But an Object might not be a Number.

Thus, we have restrictions on what we can write into our list:

- Object o = new Object(); lsn.add(o); ✗
- Number n = 4; lsn.add(n); ✓
- Integer i = 5; lsn.add(i); ✓

Remember: ArrayList<Number> must hold Numbers. We cannot add `o` because an Object cannot be cast into a Number. But we can add `i` because an Integer can be cast into a Number.

Typing and Generics

- What types can you put here?
  - **List<? super Number> lsn = new ArrayList<_________>();**
    - Object
    - Number
  - Number *is* an Object. But an Object might not be a Number.

Thus, we have restrictions on what we can read from our list:

- Object o = lsn.get(0); ✓
- Number n = lsn.get(0); ✗
- Integer i = lsn.get(0); ✗

Since **`lsn`** *could* be an ArrayList<Object>, we can only pull Objects out of here.

Typing and Generics

- <?> is a wildcard for unknown
  - Upper bounded wildcard: **`<? extends`** **______>** (subclass)
    - Safe to read from: result will be the type after **`extends`**
    - Illegal to write into (no calls to add!) because we can’t guarantee type safety.
- What types can you put here?
  - **List<? extends Number> lei = new ArrayList<________>();**
    - Number
    - Integer, Float, Double, Long…
    - Some other class that extends Integer.
    - Some other class that extends the class that extended Integer… (infinitely downwards)
  - Anything that extends Number will still be a Number. But, Number might not be an Integer, or any of its subclasses.

Typing and Generics

- What types can you put here?
  - **List<? extends Number> lei = new ArrayList<_________>();**
    - Number
    - Integer, Float, Double, Long…
    - Some other class that extends Integer.
    - Some other class that extends the class that extended Integer… (infinitely downwards)
  - Anything that extends Number will still be a Number. But, Number might not be an Integer, or any of its subclasses.

What can we write/add to the list?

- Object o = new Object();   len.add(o); ❌
- Number n = 4;   len.add(n); ❌
- Integer i = 5;   len.add(i); ❌

We cannot add anything because there is no lower bound on the actual type of the List.

Typing and Generics

- What types can you put here?
  - **List<? extends Number> lei = new ArrayList<_________>();**
    - Number
    - Integer, Float, Double, Long…
    - Some other class that extends Integer.
    - Some other class that extends the class that extended Integer… (infinitely downwards)
  - Anything that extends Number will still be a Number. But, Number might not be an Integer, or any of its subclasses.

  - Object o = len.get(0); ✓
  - Number n = len.get(0);✓
  - Integer i = len.get(0); ✗

When we retrieve/read an element, it must be of type Number. A Number is an Object, but a Number might not be an Integer.

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<CSEStudent> lcse;

List<? extends CSEStudent> lecse;

Student scholar;

CSEStudent hacker;
```

```
ls = lcse;

les.add(scholar);

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

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<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
Student scholar;
CSEStudent hacker;
```

```
ls = lcse;                X
les.add(scholar);
hacker = lecse.get(0);
```

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<CSEStudent> lcse;

List<? extends CSEStudent> lecse;

Student scholar;

CSEStudent hacker;
```

ls = lcse; X

les.add(scholar); X

hacker = lecse.get(0);

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<CSEStudent> lcse;
List<? extends CSEStudent> lecse;
Student scholar;
CSEStudent hacker;
```

ls = lcse; X

les.add(scholar); X

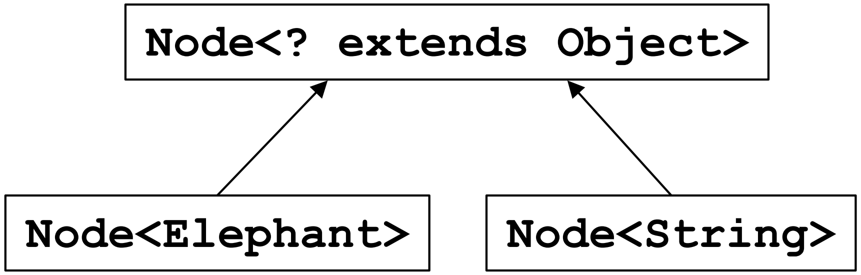
hacker = lecse.get(0); 🙂

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



More Generics

public class Box<E> { }

- **`Integer`** is a subtype of **`Number`**.

- Covariant: **`Box<Integer>`** is a subtype of **`Box<Number>`**
- Contravariant: **`Box<Number>`** is a subtype of **`Box<Integer>`**
- Invariant: neither is a subtype of the other.
  - i.e. **`Box<Number>`** and **`Box<Integer>`** are inconvertible types.

- In Java, generics are invariant.

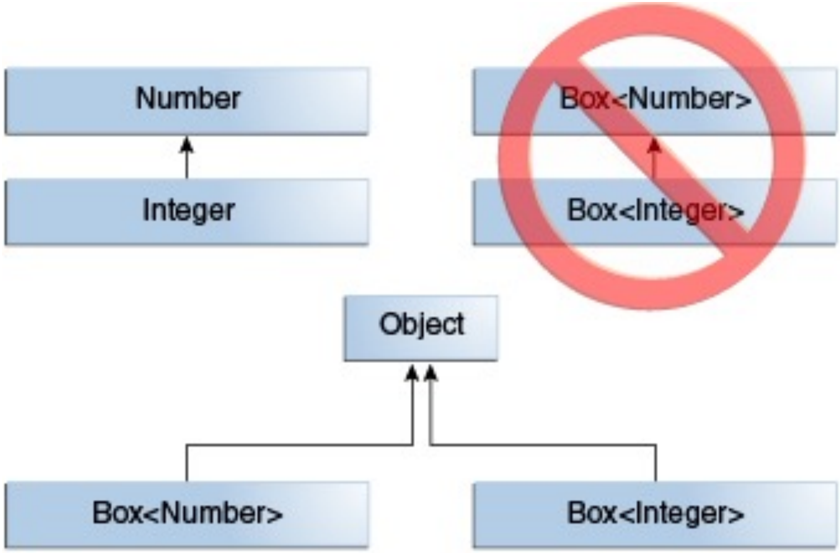
More Generics

class Box<E> { }

Is this legal?

```
public Box<Number> mystery() {
    return new Box<Integer>();
}
```

Nope. **`Box<Integer>`** is not a subtype of **`Box<Number>`**, even though **`Integer`** is a subtype of **`Number`**.

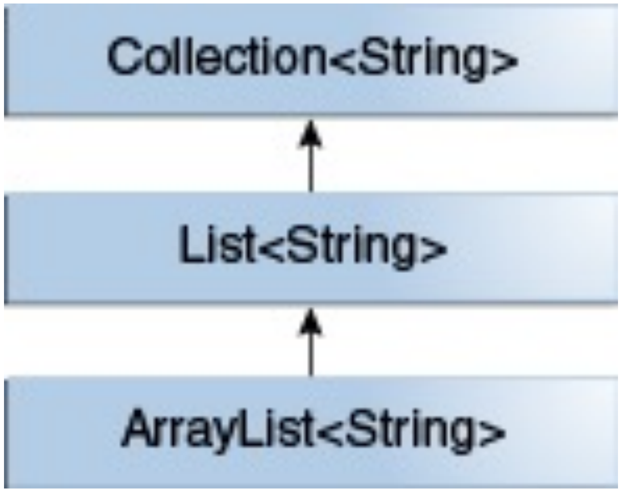


More Generics

Is this legal?

```
public List<String> mystery() {
    return new ArrayList<String>();
}
```

Yep. This is fine. As long as the type argument does not change, subtyping is preserved.



Subclasses & Overriding

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

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

Overriding a method:

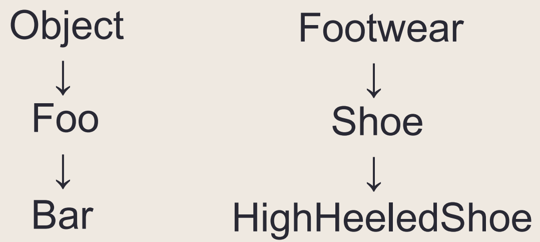
- A method overrides a superclass method only if it has the same name and exact same argument types
  - **Covariant** return types (must be a subtype of the original)

Overloading a method:

- A method overloads a method only if it has the same name and different argument types than another existing method

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) { ... }**

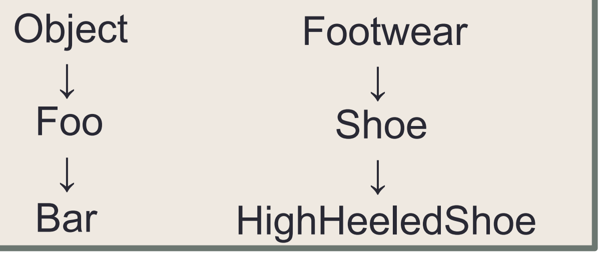
```
class Foo extends Object {
         Shoe m(Shoe x, Shoe y){ ... }
}

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

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

Method Declarations in Bar

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



Object
↓
Foo
↓
Bar

Footwear
↓
Shoe
↓
HighHeeledShoe

- **FootWear m(Shoe x, Shoe y) { ... }**

  type-error

- **HighHeeledShoe m(Shoe x, Shoe y) { ... }**

  overriding

- **Shoe m(FootWear x, FootWear y) { ... }**

  overloading

- **Shoe m(HighHeeledShoe x, HighHeeledShoe y) { ... }**

  overloading

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

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

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

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

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

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

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

```
flap flap!
quack!
```

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

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

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

```
flap flap!
quack!
```

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

```
squeak!
paddle!
squeak!
```

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

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 Duck();
b.move(42);

flap flap!
quack!

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

squeak!
paddle!
squeak!

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

Compiler error: no swim method in class Duck

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

- 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…
  - Remove the addNode/addEdge functionality from your Graph class (throw an UnsupportedOperationException when those methods are called), in order to create an UnmodifiableGraph?

  - 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…
  - Remove the addNode/addEdge functionality from your Graph class (throw an UnsupportedOperationException when those methods are called), in order to create an UnmodifiableGraph?
    - **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**

Design Patterns

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

What pattern is shown in each example?

```
Pizza pie = genericPizza.clone();
```

```
Pizza pie = new PizzaMaker()
                 .cheese(“mozzarella”)
                 .toppings(“mushrooms”)
                 .size(size.LARGE)
                 .make();
```

```
Pizza pie = getNewPizza();
```

Design Patterns

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

What pattern is shown in each example?

```
Pizza pie = genericPizza.clone();
```

prototype

```
Pizza pie = new PizzaMaker()
                 .cheese("mozzarella")
                 .toppings("mushrooms")
                 .size(size.LARGE)
                 .make();
```

```
Pizza pie = getNewPizza();
```

Design Patterns

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

What pattern is shown in each example?

```
Pizza pie = genericPizza.clone();
```

prototype

```
Pizza pie = new PizzaMaker()
                 .cheese(“mozzarella”)
                 .toppings(“mushrooms”)
                 .size(size.LARGE)
                 .make();
```

builder

```
Pizza pie = getNewPizza();
```

Design Patterns

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

What pattern is shown in each example?

```
Pizza pie = genericPizza.clone();
```

prototype

```
Pizza pie = new PizzaMaker()
                .cheese(“mozzarella”)
                .toppings(“mushrooms”)
                .size(size.LARGE)
                .make();
```

builder

```
Pizza pie = getNewPizza();
```

factory

That’s it for the review!

- There are additional practice problems at the end of these slides (with answers!)
  - Good source to review as you begin your studying

- Reminders:
  - HW9 due Saturday
  - Please demo your applications in lecture tomorrow!
  - Review session on Monday
  - Final on Tuesday

Thanks for the great quarter!

Additional Practice

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 = 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 = 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); ☺

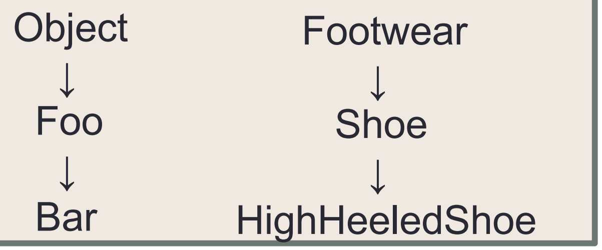
scholar = lscse.get(0); X

hacker = lecse.get(0); ☺

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



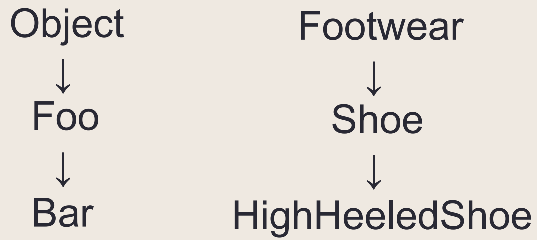
- **FootWear m(Shoe x, Shoe y) { ... }**
- **Shoe m(Shoe q, Shoe z) { ... }**
- **HighHeeledShoe m(Shoe x, Shoe y) { ... }**
- **Shoe m(FootWear x, HighHeeledShoe y) { ... }**
- **Shoe m(FootWear x, FootWear y) { ... }**
- **Shoe m(Shoe x, Shoe y) { ... }**
- **Shoe m(HighHeeledShoe x, HighHeeledShoe 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() { spea
 public void swim() { Syst                        ); }
}
```

Compile error: cannot create instances of an abstract class.

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!"); }
}
```

chirp!
chirp!

Bird
b.mo

Bird b = ne 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 Duck();
b.move(42);

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

Duck();

Duck();
donald.move();

flap flap!
quack!

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!"); }
}
```

Bi
b.

squeak!
paddle!
squeak!

Bi
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 RubberDuck();
b.move(3);

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

Compile error: no swim method in class Duck

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 RubberDuck();
b.move(3);

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

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

squeak!
paddle!

Design Patterns

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

Answer the following design pattern questions:

- You need a generic graph class, similar to HW5, but the edges must not have labels. What structural pattern best fits?

- Your web app, Campus Paths, is now very popular; to boost performance, you want the model to cache frequent path queries. Which structural pattern best fits the task?

- If a class's objects never change once initialized but must be initialized incrementally, which creational pattern would let you make the class's objects immutable?

Design Patterns

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

Answer the following design pattern questions:

- You need a generic graph class, similar to HW5, but the edges must not have labels. What structural pattern best fits? **Adaptor**

- Your web app, Campus Paths, is now very popular; to boost performance, you want the model to cache frequent path queries. Which structural pattern best fits the task?

- If a class's objects never change once initialized but must be initialized incrementally, which creational pattern would let you make the class's objects immutable?

Design Patterns

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

Answer the following design pattern questions:

- You need a generic graph class, similar to HW5, but the edges must not have labels. What structural pattern best fits? **Adaptor**
- Your web app, Campus Paths, is now very popular; to boost performance, you want the model to cache frequent path queries. Which structural pattern best fits the task? **Decorator**
- If a class's objects never change once initialized but must be initialized incrementally, which creational pattern would let you make the class's objects immutable?

Design Patterns

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

Answer the following design pattern questions:

- You need a generic graph class, similar to HW5, but the edges must not have labels. What structural pattern best fits? **Adaptor**
- Your web app, Campus Paths, is now very popular; to boost performance, you want the model to cache frequent path queries. Which structural pattern best fits the task? **Decorator**
- If a class's objects never change once initialized but must be initialized incrementally, which creational pattern would let you make the class's objects immutable? **Builder**