Review

Subjects Covered

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

Stronger vs Weaker (one more time!)

- Requires more about inputs?

- Promises more about behavior?

Stronger vs Weaker (one more time!)

- Requires more about inputs?

weaker

- Promises more about behavior?

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
```

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

```
@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

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:
  1. catch the exception
  2. 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

```
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

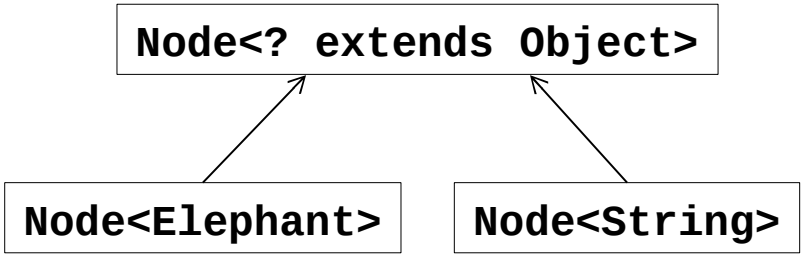
hacker = lecse.get(0); 

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



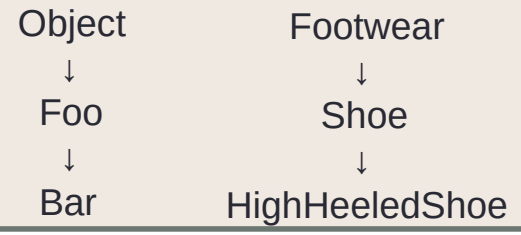
Subclasses & Overriding

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

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

Method Declarations in Bar

- 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!"); }
}
```

```
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 mov                                      }
 public void swi                                     n("paddle!"); }
}
```

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!"); }
}
```

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

```
Chirp!
Chirp!
```

```
Duck();
```

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

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

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

```
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("p
}
```

flap flap!
quack!

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

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

```
RubberDuck();
```

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

squeak!
paddle!
squeak!

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.ou
}
```

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!"); }
}
```

squeak!
paddle!

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

```
Bird b = ne
b.move(42
```

```
ld = new RubberDuck();
m();
```

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

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

```
Duck 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:
  1. GUIs
  2. Web servers

- Where is the event loop in Spark Java?
  - it is created behind the scenes