CSE 331
Software Design & Implementation

Winter 2026

Section 9 – Polymorphism

Administrivia

- HW8 released tonight - **Due @ 11:59pm next Wednesday**
- Next section is **final review**
- The final is on **3/17** (more details coming next week)

Subtypes

Strength

- ADT **B** is a *subtype* of **A** if:
  - B has all the methods of A
  - Each method of B has a *stronger spec* than A's corresponding methods
- If B is a subtype of A, then B is *stronger* than A
  - A is a *supertype* of B

* not enforced by Java, but very important practice to **only define subclasses that are also subtypes**!

- Ex: Every Unicorn is a Pony

```
class Pony {...}
class Unicorn extends Pony {...}
```

Type Bounds

Upper Bound: extends

- Type argument passed in must be the same type or a subclass

Lower Bound: super

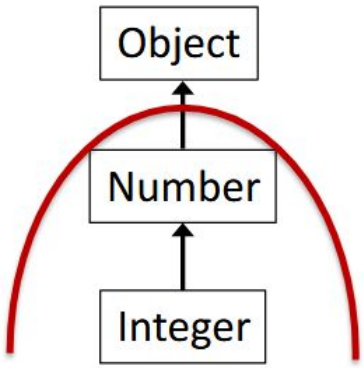
- Type argument passed in must be the same type or a superclass

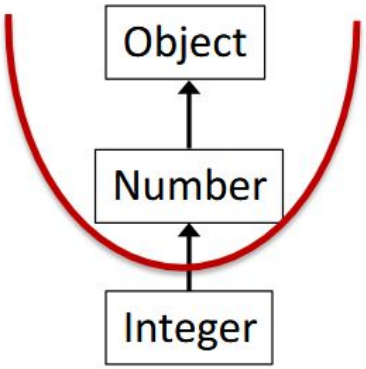
Upper Bound

```
? extends Number
```

Lower bound

```
? super Number
```



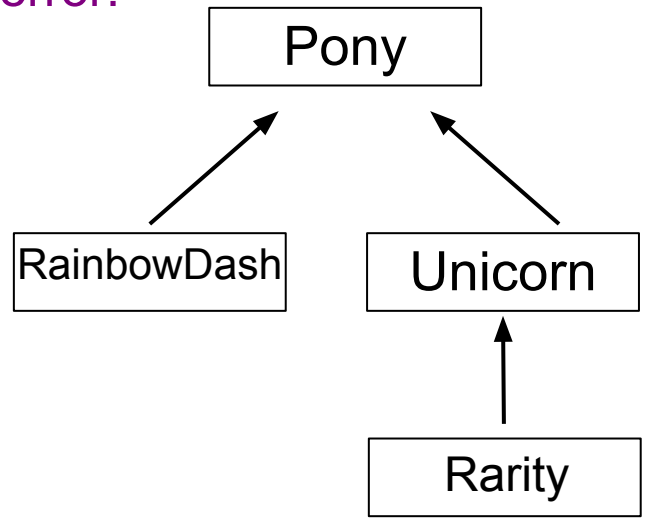


Type Constraints: Extends

Ex:

```
List<? extends Unicorn> unicorns = new ArrayList<Unicorn>()

unicorns = new ArrayList<Rarity> // okay!
unicorns = new ArrayList<RainbowDash> // error!
```

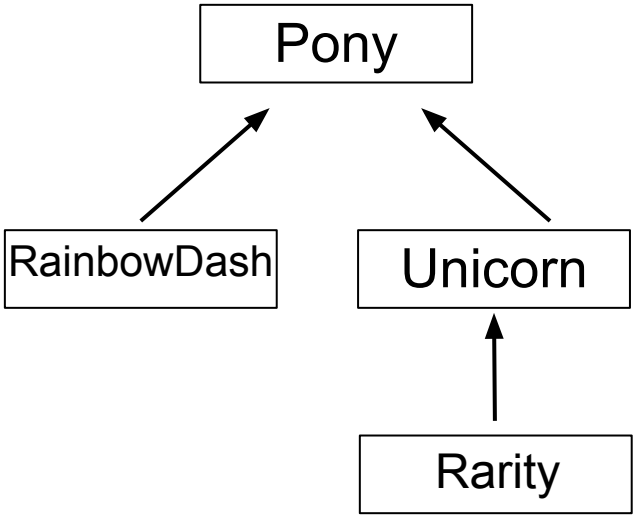


Type Constraints: Super

Ex:

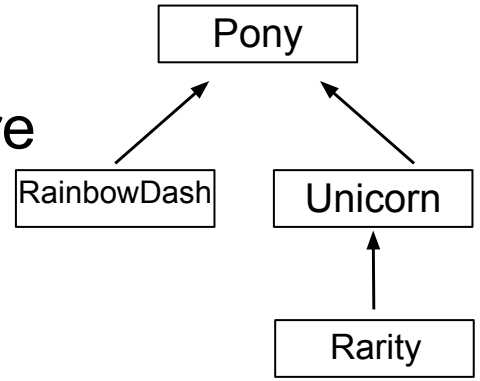
```
List<? super Unicorn> ponies = new Arraylist<Pony>()

ponies = new ArrayList<Rarity> // error!
ponies = new ArrayList<Pony> // ok!
```



Type Constraints on Methods

Code can only use methods from type bounds
- Can only call methods *guaranteed* to be there



Ex:

```
class PinkyPie<E extends Pony>{
    public string magic(E arg) {
        return arg.magic();  // error! (Pony not guaranteed to magic())
    }
}

class Twilight<E extends Unicorn>{
    public string magic(E arg) {
        return arg.magic();  // okay! (Unicorn is guaranteed to magic())
    }
}
```

Generics

Throughout this entire class we've been dealing with Lists of numbers, booleans, and pairs

- type List := nil | cons(x: $\mathbb{Z}$, L: List)
- type BList := nil | cons(f: $\mathbb{B}$, L: BList)
- type PList := nil | cons(p: $\mathbb{Z} \times \mathbb{Z}$, L: PList)

What if we wanted to create a definition for *any* list?

type List<A> := nil | cons(x: A, L: List<A>)

Ex:

```
class FastLastList<T> implements List<T> {
    private List<T> list;
    private T last;
}
```

Generic Methods

Ex: Summing a List of integers or doubles:

```
static <T extends Number> double sum(List<T> list) {
    double result = 0;
    for (T n : list) {
        result += n.doubleValue();    // Legal since T extends
    }                                    Number
    return result;
}
```

Wildcards: ?

?: Concise way of writing some generics

- "**?** **extends** E": **?** is an *anonymous* subclass of E
- "**?** **super** E": **?** is an *anonymous* superclass of E
- "**?**" is an anonymous subclass of *Object*
- Each "**?**" is independent from each other
- Allows for flexible input types

? vs Object:

- List<?> allows for *Object*, but also Integer, String, etc…
- Cannot pass List<Integer> for List<Object>
- Can pass List<Integer> as List<?>

T vs ?

- Use **T** to explicitly define, control, and reuse a given type
- Use **?** for dynamic, non-reusable input types where you don't need to know the exact type

Ex:

Both of these functions are equivalent:

- <**T1, T2**> **void** foo(List<**T1**> list, List<**T2**> list2) {...}
- **void** foo(List<**?**> list, List<**?**> list2) {...}

If you want both lists to have the same type, you *must* use **T**:

- <**T**> **void** foo(List<**T**> list, List<**T**> list2) {...}

Generic Method Declarations

- We can compare generic method declarations similarly to how we compare specifications:
    - Parameters = preconditions
    - Return types = postconditions

Strength rules:

- A method is **stronger** if:
    - it accepts *more* argument types (weaker precondition)
    - it returns a *more* specific type (stronger postcondition)
- A method is **weaker** if:
    - it accepts *fewer* argument types (stronger precondition)
    - it returns a *less* specific type (weaker postcondition)

Generics Control Strength

When comparing declarations, ask:

- Which accepts more possible argument types?
- Which enforces stronger type relationships?
- Does one strictly accept more valid calls than the other?

If yes, it is stronger. (weaker precondition)

Guidelines:

- **extends** → widens acceptable inputs → *weaker* precondition → *stronger* method
- **super**→ widens acceptable inputs → *weaker* precondition → *stronger* method
- **<T>** → ties types together → *stronger* type relationship
- **?** → each ? is independent → *weaker* type relationship

If neither allows strictly more calls or has stronger guarantees → **incomparable**