CS2030
Lab 5

AY24/25 Sem 2, Week 9

Fadhil Peer Mohamed <f_p_m@u.nus.edu>
Pang Yang Yi <pang.yy@u.nus.edu>

Deadlines

Timeline

WEEK	**LAB**	**DUE**
9	5	*none! :)*
10	Well-being Day	5 - 27 Mar 2359 (Thur)
11	Mock PA#2	Project - 6 Apr 2359 (Sun)
12	PA#2	Mock PA#2 - 10 Apr 2359 PA#2 Moderation - 20 Apr 2359 (Wk13 Sunday) (updated!)

Recap

Bounded Wildcards

What can Box store?

Box<? **extends** T> Upper bounded wildcard

Box<? **super** T> Lower bounded wildcard

Box<?> Unbounded wildcard

Bounded Wildcards

Imagine you have two different kind of lists:

```
List<Cat> catList = List.of(new Cat("Black Cat"),
                            new Cat("Orange Cat"), new Cat("White Cat"));
List<Dog> dogList = List.of(new Dog("Chihuahua"),
                            new Dog("Dachshund"), new Dog("Bulldog"));
```

Both Cat and Dog extends Animal.

Bounded Wildcards

Will the following method work?

```
void print(List<Animal> animalList)
```

No, because generics are invariant, i.e. even if Cat is a subtype of Animal, List<Cat> is not a subtype of List<Animal>. To create a method that accepts different kind of Lists of animals, we will need wildcard:

```
void print(List<? extends Animal> animalList)
```

Bonus Question: Why extends instead of super?

PECS

PECS: Producer **extends**, Consumer **super**

Think of “data flowing out” for Producers (Producer *produces*, so data flowing out)

And “data flowing in” for Consumers (Consumer *consumes*, so data flowing in)

PECS

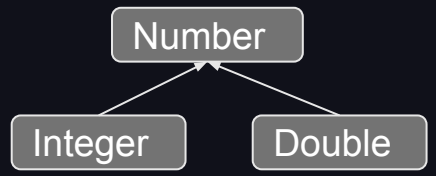
PECS: Producer **extends**, Consumer **super**

Underlying principle on which to use:
<? **extends** T> if you want to read from a collection (producer)
<? **super** T> if you want to add to a collection <? **super** T>

Use <T> if you want to read from and add to a collection

Producer Extends

Given this inheritance diagram, consider this List of Numbers:



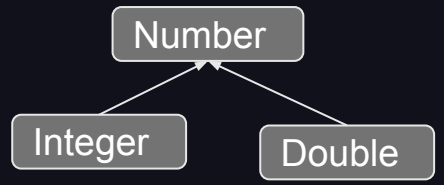

```
List<? extends Number> numList;
```

What are the possible types that the numList contains?

Producer Extends

```
List<? extends Number> numList = new ArrayList<Number>();
List<? extends Number> numList = new ArrayList<Integer>();
List<? extends Number> numList = new ArrayList<Double>();
```

These assignments are all valid, since Integer and Double extend Number



Producer Extends

Now, assume we want to read from the numList. Data is flowing out of numList, hence we use a Producer - <? **extends** T>

Let's look at our options for variable assignment:

```
Integer i = numList.get(x);
Double  d = numList.get(x);
Number  n = numList.get(x);
```

Which one of these is a valid assignment?

Producer Extends

Without knowledge of what exactly numList holds:

```
Integer i = numList.get(x);
```

You cannot read an Integer because numList could be pointing to a List<Double>

```
List<? extends Number> numList = new ArrayList<Double>();
```

Producer Extends

Without knowledge of what exactly numList holds:

```
Double  d  = numList.get(x);
```

You cannot read a Double because numList could be pointing to a List<Integer>

```
List<? extends Number> numList = new ArrayList<Integer>();
```

Producer Extends

Without knowledge of what exactly numList holds:

```
Number n  = numList.get(x);
```

This is the only “safe” assignment, since regardless of what the numList contains, it will be a subclass of Number

Producer Extends

When writing to numList:

- You **cannot** add an Integer because numList could be pointing to a List<Double>
- You **cannot** add a Double because numList could be pointing to a List<Integer>
- You **cannot** add a Number because numList could be pointing to a List<Integer>

```
List<? extends Number> numList = new ArrayList<Number>();
List<? extends Number> numList = new ArrayList<Integer>();
List<? extends Number> numList = new ArrayList<Double>();
```

Producer Extends

You can't add any object to List<? **extends** T> because you can't guarantee what kind of List it is really pointing to, so you can't guarantee that the object is allowed in that List. The only "guarantee" is that you can only read from it and you'll get a T or subclass of T.

Conclusion: By using **extends**, only Numbers (or subclasses of Number) can be read from numList (therefore numList is a producer of Numbers)

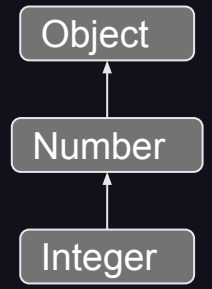
```
List<? extends Number> numList = new ArrayList<Number>();
List<? extends Number> numList = new ArrayList<Integer>();
List<? extends Number> numList = new ArrayList<Double>();
```

Consumer Super

We reconstruct the numList with the following possibilities:

```
List<? super Integer> numList = new ArrayList<Integer>();
List<? super Integer> numList = new ArrayList<Number>();
List<? super Integer> numList = new ArrayList<Object>();
```

These assignments are all valid, since Number and Object are superclasses of Integer



Consumer Super

Now, what happens when we try to read out of numList?

Let's look at our options for variable assignment:

```
Integer i = numList.get(x);
Number  n = numList.get(x);
Object  o = numList.get(x);
```

Which one of these is a valid assignment?

Consumer Super

Without knowledge of what exactly numList holds:

```
Integer i = numList.get(x);
```

You cannot read an Integer because numList could be pointing to a List<Number> or List<Object>

```
List<? super Integer> numList = new ArrayList<Integer>();
List<? super Integer> numList = new ArrayList<Number>();
List<? super Integer> numList = new ArrayList<Object>();
```

Consumer Super

Without knowledge of what exactly numList holds:

```
Number n = numList.get(x);
```

You cannot read a Number because numList could be pointing to a List<Object>

```
List<? super Integer> numList = new ArrayList<Integer>();
List<? super Integer> numList = new ArrayList<Number>();
List<? super Integer> numList = new ArrayList<Object>();
```

Consumer Super

Without knowledge of what exactly numList holds:

```
Object o = numList.get(x);
```

You can only read an Object but the subclass (if any) is unknown (redundant to read an Object; every class is a subclass of Object)

```
List<? super Integer> numList = new ArrayList<Integer>();
List<? super Integer> numList = new ArrayList<Number>();
List<? super Integer> numList = new ArrayList<Object>();
```

Consumer Super

When writing to numList, data is flowing into numList, hence we use a Consumer - <? **super** T>:

- You can add an Integer (allowed by all 3 lists)
- You can add an instance of a subclass of Integer (allowed by all 3 lists)
- You cannot add a Double, Number or Object because numList might point to a List<Integer>

```
List<? super Integer> numList = new ArrayList<Integer>();
List<? super Integer> numList = new ArrayList<Number>();
List<? super Integer> numList = new ArrayList<Object>();
```

Consumer Super

You can't read from a List<? **super** T> because you do not know what kind of List it points to (unless you assign the item to an Object instance; redundant as mentioned before). You can only add items of type T or a subclass of T to the list.

Conclusion: By using **super**, only Integers (or subclasses of Integer) can be added to numList (therefore numList is a consumer of Integers)

```
List<? super Integer> numList = new ArrayList<Integer>();
List<? super Integer> numList = new ArrayList<Number>();
List<? super Integer> numList = new ArrayList<Object>();
```

PECS: Function<T,R>

What does T and R correspond to? What do they mean?

Method Summary

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method	Description
R	apply(T t)	Applies this function to the given argument.

If I were to create a method that takes in a function, which bounded wildcard should I use?

1. Function<? extends T,? extends R>
2. Function<? super T,? extends R>
3. Function<? extends T,? super R>

Lab 5

Optional - orElseThrow

We will be looking at the orElseThrow method for today's lab

`<X extends Throwable>` `T`	`orElseThrow(Supplier<? extends X> exceptionSupplier)` Return the contained value, if present, otherwise throw an exception to be created by the provided supplier.

Throws an Exception to be generated by the Supplier if the Optional is empty

Optional - orElseThrow

Using your project as an example…

```
shop.findServer(cust)
    .orElseThrow(() -> new Exception("No Server found!"))
```

Task Overview

We want to log the changes that happen to values while they are operated upon, as a way to emulate debugging statements

To do this, we are to define a generic Log<T> class

Hints

Make use of the orElseThrow and filter methods to get your desired behaviour!

Deadline: **27 Mar (Thurs) 2359**