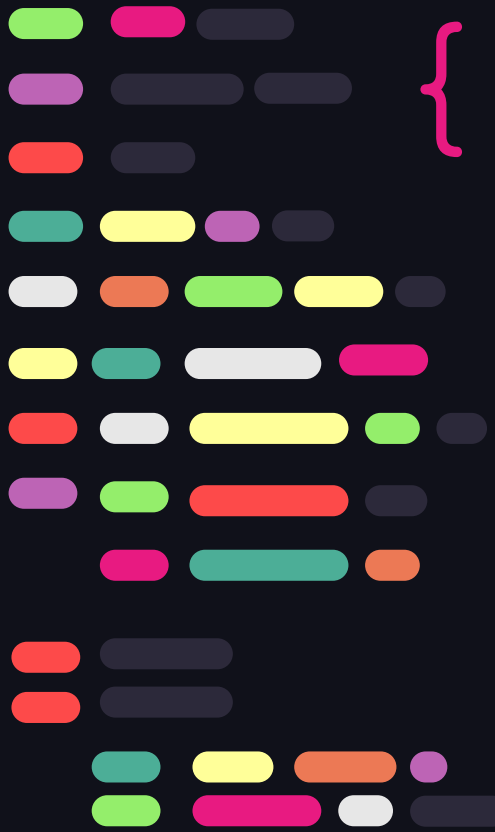


CS2030
Lab 5

AY25/26 Sem 1, Week 10

Fadhil Peer Mohamed <f_p_m@u.nus.edu>
Abner Then <abner.then@u.nus.edu>

Project

There is no need for a queue data structure in Server

- If a customer arrives first, he will be earlier in the queue
- Events belonging to customers with the earliest arrival time occurs first, if the events happen at the same time
- This means the events will always occur in the order of customers in the queue
- When a server is available, the first customer in the queue will be served, and the rest will continue to wait since the server is now serving the customer that was in front

Project

The deadline has been extended till **Sunday 26 Oct 2359**

Everyone can resubmit, including those who got A but want to improve their design

If there are unused methods leftover from previous labs, delete them to make grading easier. Also delete **commented blocks of code**

Recap

Bounded Wildcards

```
public record Box<T>(T t);
```

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:

Both Cat and Dog extend Animal.

```
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"));
```

Bounded Wildcards

Will the following method accept the prior lists as input?

```
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**

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>

PECS

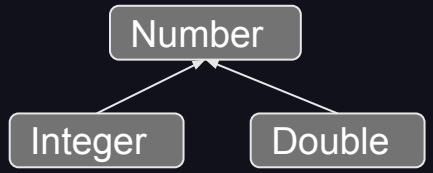
PECS: Producer **extends**, Consumer **super**

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)

Producer Extends

Given this inheritance diagram, consider this List of Numbers:



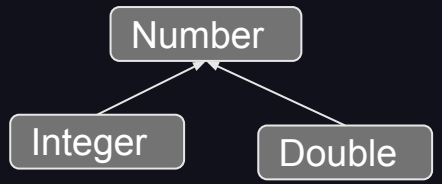

```
List<? extends Number> numList;
```

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 the data is of type - <? **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 objects List really holds.

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)

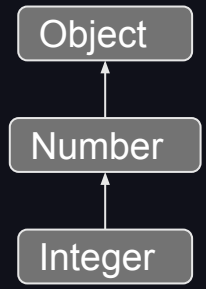
```
List<? extends Number> numList = new ArrayList<Number>();
List<? extends Number> numList = new ArrayList<Integer>();
List<? extends Number> numList = new ArrayList<Double>();
```

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>`**

When to use

Adding wildcards only limits what you can do with your lists

- List<? super T>: can add subclasses of T, cannot read
- List<? extends T>: can only read T, cannot add
- List<T>: can add subclasses of T, can read superclasses of T

No point restricting the operations that you can perform by adding wildcards if you do not need the wildcards

When to use

As a guideline, the only place you will usually use it is when declaring function signatures

Consider a Maybe<Integer>. We can map it using a few different functions

- Function<Integer, Integer> f1 = x -> x + 1
- Function<Number, Integer> f2 = x -> x.hashCode()
- Function<Object, String> f3 = x -> x.toString()

Are all these valid functions to pass to map?

When to use

- `Function<Integer, Integer> f1 = x -> x + 1`
- `Function<Number, Integer> f2 = x -> x.hashCode()`
- `Function<Object, String> f3 = x -> x.toString()`

If map takes in a Function<T, R>, and T is Integer, then only f1 works, although all should work. Therefore we need the signature Function<? **super** T, R>

```
jshell>  /list

   1 : class Maybe<T> {
           T val;

           Maybe(T val) {
               this.val = val;
           }

           <R> Maybe<R> map(Function<T,R> mapper) {
               return new Maybe<R>(mapper.apply(val));
           }

           public String toString() {
               return String.format("Maybe[%s]", val);
           }
       }
   2 : Function<Integer, Integer> f1 = x -> x + 1;
   3 : Function<Number, Integer> f2 = x -> x.hashCode();
   4 : Function<Object, String> f3 = x -> x.toString();
   5 : Maybe<Integer> maybe = new Maybe<>(1);

jshell> maybe.map(f1)
$6 ==> Maybe[2]

jshell> maybe.map(f2)
|  Error:
|  method map in class Maybe<T> cannot be applied to given types;
|    required: java.util.function.Function<java.lang.Integer,R>
|    found:    java.util.function.Function<java.lang.Number,java.lang.Integer>
|    reason: cannot infer type-variable(s) R
|      (argument mismatch; java.util.function.Function<java.lang.Number,java.lang.Integer> cannot be converted to java.util.function.Function<java.lang.Integer,R>)
|  maybe.map(f2)
|  ^-------^

jshell> maybe.map(f3)
|  Error:
|  method map in class Maybe<T> cannot be applied to given types;
|    required: java.util.function.Function<java.lang.Integer,R>
|    found:    java.util.function.Function<java.lang.Object,java.lang.String>
|    reason: cannot infer type-variable(s) R
|      (argument mismatch; java.util.function.Function<java.lang.Object,java.lang.String> cannot be converted to java.util.function.Function<java.lang.Integer,R>)
|  maybe.map(f3)
|  ^-------^
```

```
jshell>  /list

   1 : class Maybe<T> {
           T val;

           Maybe(T val) {
               this.val = val;
           }

           <R> Maybe<R> map(Function<? super T,R> mapper) {
               return new Maybe<R>(mapper.apply(val));
           }

           public String toString() {
               return String.format("Maybe[%s]", val);
           }
       }
   2 : Function<Integer, Integer> f1 = x -> x + 1;
   3 : Function<Number, Integer> f2 = x -> x.hashCode();
   4 : Function<Object, String> f3 = x -> x.toString();
   5 : Maybe<Integer> maybe = new Maybe<>(1);

jshell> maybe.map(f1)
$6 ==> Maybe[2]

jshell> maybe.map(f2)
$7 ==> Maybe[1]

jshell> maybe.map(f3)
$8 ==> Maybe[1]
```

When to use

```
Function<Integer, Integer> f1 = x -> x + 1
Maybe<Integer> maybe = Maybe.of(1)
 -  Maybe<Integer> m1 = maybe.map(f1)
 -  Maybe<Object> m2 = maybe.map(f1)
```

If map’s signature is Function<T, R>, m1 will work but m2 will not. In the case of m2, R is Object because map returns a Maybe<R>, but there is an issue because f1 is Function<Integer, Integer> and Integer != Object

When to use

```
Function<Integer, Integer> f1 = x -> x + 1
Maybe<Integer> maybe = Maybe.of(1)
 -  Maybe<Integer> m1 = maybe.map(f1)
 -  Maybe<Object> m2 = maybe.map(f1)
```

If it is Function<T, ? **extends** R>, then R can be Object, and passing in f1 will work because Integer extends Object. Therefore the use of wildcards is to allow methods like map to accept a variety of inputs

When to use

```
jshell>  /list

   1 : class Maybe<T> {
           T val;

           Maybe(T val) {
               this.val = val;
           }

           <R> Maybe<R> map(Function<T,R> mapper) {
               return new Maybe<R>(mapper.apply(val));
           }

           public String toString() {
               return String.format("Maybe[%s]", val);
           }
       }
   2 : Function<Integer, Integer> f1 = x -> x + 1;
   3 : Maybe<Integer> maybe = new Maybe<>(1);

jshell> Maybe<Integer> m1 = maybe.map(f1)
m1 ==> Maybe[2]

jshell> Maybe<Object> m2 = maybe.map(f1)
|  Error:
|  incompatible types: inference variable R has incompatible equality constraints java.lang.Object,java.lang.Integer
|  Maybe<Object> m2 = maybe.map(f1);
|                     ^-----------^
```

When to use

```
jshell>  /list

   1 : class Maybe<T> {
           T val;

           Maybe(T val) {
               this.val = val;
           }

           <R> Maybe<R> map(Function<T,? extends R> mapper) {
               return new Maybe<R>(mapper.apply(val));
           }

           public String toString() {
               return String.format("Maybe[%s]", val);
           }
       }
   2 : Function<Integer, Integer> f1 = x -> x + 1;
   3 : Maybe<Integer> maybe = new Maybe<>(1);

jshell> Maybe<Integer> m1 = maybe.map(f1)
m1 ==> Maybe[2]

jshell> Maybe<Object> m2 = maybe.map(f1)
m2 ==> Maybe[2]
```

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: **30 Oct (Thurs) 2359**