Welcome to CS2030S Lab 6!

15 October 2021 [16A]

Please login to your pe node once it's 4pm.

Project

- Please try to start early?
- Deadline: 24 Nov 2021 23:59

COVID-19

- Please please if you have any symptoms, please contact us and don't force yourself to attend physical lab!

Submission Statistics

Labs	**Submitted**	**Not Submitted**	**A**
1🟢	17	0	17🤩
2🟡	17	0	16
3🟡	16	1	12
4🟡	17	0	15
5🟡	17	0	9
Project😂	1	**16**	0

Bounded Wildcards

- `Box<? extends T>` // Box can have any subclass of T
- `Box<? super T>` // Box can have any superclass of T
- `Box<?>` // Synonymous with Box<? extends Object>

PECS

- PECS: Producer extends, Consumer super
- Underlying principle: Use `<? extends T>` if you want get from a collection (producer), and `<? super T>` if you want to add to a collection `<? super T>`
- Use <T> if you want a collection to support BOTH get and add!

Producer Extends

```
List<? extends Number> foo3 = new ArrayList<Number>();  // Number "extends" Number (in this context)
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>();  // Double extends Number
```

- The above assignments are all valid
- Getting from foo3 (must be legal for **all** possible foo3 assignments):
  - You can get a `Number` because any of the lists that could be assigned to foo3 contain a `Number` or a subclass of `Number`

```
List<? extends Number> foo3 = new ArrayList<Number>();  // Number "extends" Number (in this context)
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>();  // Double extends Number
```

- Adding to foo3 (must be legal for **all** possible foo3 assignments):
  - **You cannot add anything into it**
  - Cannot add `Integer` because foo3 could be pointing to `List<Double>`
  - Cannot add `Double` because foo3 could be pointing to a `List<Integer>`
  - Cannot add `Number` because foo3 could be pointing to a

```java
List<? extends Number> foo3 = new ArrayList<Number>();  // Number "extends" Number (in this context)
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>();  // Double extends Number
```

- 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 get from it and you'll get a T or subclass of T.

- Conclusion: **Don't use** `extends` **if you want to add to a collection**

Consumer Super

```
List<? super Integer> foo3 = new ArrayList<Integer>();  // Integer is a "superclass" of Integer (in this context)
List<? super Integer> foo3 = new ArrayList<Number>();   // Number is a superclass of Integer
List<? super Integer> foo3 = new ArrayList<Object>();   // Object is a superclass of Integer
```

- The above assignments are all valid.
- Getting from foo3 (must be legal for **all** possible foo3 assignments):
  - You cannot get an `Integer` because foo3 could be pointing to a `List<Number>` or `List<Object>`

```
List<? super Integer> foo3 = new ArrayList<Integer>();  // Integer is a "superclass" of Integer (in this context)
List<? super Integer> foo3 = new ArrayList<Number>();   // Number is a superclass of Integer
List<? super Integer> foo3 = new ArrayList<Object>();   // Object is a superclass of Integer
```

- Adding to foo3 (must be legal for **all** possible foo3 assignments):
  - 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` because foo3 might point to a List<Integer>
  - You cannot add a `Number` because foo3 might point to a

```
List<? super Integer> foo3 = new ArrayList<Integer>();  // Integer is a "superclass" of Integer (in this context)
List<? super Integer> foo3 = new ArrayList<Number>();   // Number is a superclass of Integer
List<? super Integer> foo3 = new ArrayList<Object>();   // Object is a superclass of Integer
```

- You can't get 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: **Use super to add objects into your collection**

Single Abstract Method (SAM)/Functional Interfaces

- `Function<T, R>` is a SAM interface (i.e. it is an interface containing only one abstract method for implementing classes to implement)
- Only (abstract) method in Function interface:
  ```
  java
  public R apply(T t) // Takes in a T and returns
  an R
  ```

Anonymous Classes

- You can instantiate a class without defining it proper.
- Eg. Anonymous class that implements `Function`:

```
java
Function<Integer, Integer> addOne = new
Function<>() {
@Override
public Integer apply(Integer x) {
    return Integer.valueOf(x.intValue() + 1);
}
```

Lambda Expression

- A shorter way of creating a `Function`.
  ```
  java
  Function<Integer, Integer> addOne = x -> x + 1;
  ```

- Takes in an `Integer` x and returns an Integer `x`

- How does the Java compiler know which method the lambda expression is implementing? Well, `Function` is a SAM interface, so there is only one method that it can implement (the `apply` method)

Optional

- `Optional` class is a useful abstraction to deal with **null** values
- `Optional<T>` creates an `Optional` that wraps around type `T` (e.g. `Optional<Integer>` creates an `Optional` that wraps around an `Integer`)
- Sounds familiar? (*Hint: refer to one of the recent recitation questions*)

Optional - Useful API methods

```
Optional empty();

T get();

boolean isPresent();

Optional ofNullable(T value);

void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction);

void orElseThrow(Supplier<? extends X> exceptionSupplier);

Optional map(Function<? super T, ? extends U> mapper);

Optional flatMap(Function<? super T, ? extends Optional<? extends U>> mapper);
```

Optional - PECS in context

- The `Optional` class uses PECS to support passing super/subclasses of wrapped types into methods
- For instance, `Function<? super T, ? extends U>` mapper is passed into the map method
- In context, it means that mapper can take any superclass of `T` (stored in `Optional`) as input and have any subclass of `U` as output (`U`'s type depends on the `Function` being used)

Invariance, Contravariance, Covariance

- In this case, assume that f maps the type to a data structure containing objects of that type
- In Java, **arrays are covariant**
  ```
  java
  Object[] items = new String[1]; // Ok; String
  is a subclass of Object
  ```
- On the other hand, **generics are invariant**
  ```
  java
  ```

Java Generics

- Therefore, `List<?>` is not the same as `List<Object>`
- `List<?>` is **more general** than `List<Object>` because <?> is synonymous with <? extends Object>
- `List<?>` can store instances of type `Object` or any subclass of `Object`, i.e. `List<?>` can store anything since all other classes are subclasses of `Object`

Map

- A map is a data structure which maps a key to a value.
- One key can only be mapped to one value.
- Mapping a an existing key to another value will replace the current mapping in the map.
- Think of it as a dictionary (for students familiar with Python/ JavaScript).

Sources

Covariance, Invariance and Contravariance

Differences in `?` and `Object`

Difference between `super` and `extends`