CS2030 Programming Methodology
Semester 1 2023/2024

4 & 5 October 2023
Problem Set #5 Suggested Guidance
More Java Generics

1. In the lecture, we have seen the use of the `Comparator<T>` interface with the method specification `int compare(T t1, T t2)` that returns zero if `t1` and `t2` are equal, a negative integer if `t1` is less than `t2`, or a positive integer if `t2` is less than `t1`.

```
public interface Comparator<T> { // <T> declared with class scope
    int compare(T o1, T o2);
    ...
}
```

A generic method `T max3(T a, T b, T c, Comparator<T> comp)` can be defined in JShell as shown below. The method takes in three values of type `T` as well as a `Comparator<T>`, and returns the maximum among the values.

```
jshell> <T> T max3(T a, T b, T c, Comparator<T> comp) { // <T> declared with
   ...>     T max = a;                                  // method scope
   ...>     if (comp.compare(b, max) > 0) {
   ...>         max = b;
   ...>     }
   ...>     if (comp.compare(c, max) > 0) {
   ...>         max = c;
   ...>     }
   ...>     return max;
   ...> }
|  created method max3(T,T,T,Comparator<T>)
```

(a) Demonstrate how the `max3` method can be called so as to return the maximum of three integers -1, 2 and -3.

```
jshell> class IntComparator implements Comparator<Integer> {
   ...>     public int compare(Integer i1, Integer i2) {
   ...>         return i1 - i2;
   ...>     }
   ...> }
jshell> max3(-1, 2, -3, new IntComparator());
$.. ==> 2
```

(b) Other than `Comparator<T>`, there is a similar `Comparable<T>` interface with the method specification `int compareTo(T o)`. This allows one `Comparable` object to compare itself against another `Comparable` object.

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

As an example, since `Integer` class implements `Comparable<Integer>`,

```
jshell> Integer i = 1 // 1 autoboxed to an Integer and assigned to i
i ==> 1

jshell> i.compareTo(2) // 2 autoboxed to an Integer and passed to compareTo
$.. ==> -1
```

Let's redefine the `max3` method to make use of the `Comparable` interface instead.

```
<T> T max3(T a, T b, T c) {
    T max = a;
    if (b.compareTo(max) > 0) {
        max = b;
    }
    if (c.compareTo(max) > 0) {
        max = c;
    }
    return max;
}
```

Does the above method work? What is the compilation error?

```
jshell> /open ...
|  Error:
|  cannot find symbol
|    symbol:   method compareTo(T)
|      if (b.compareTo(max) > 0) {
|          ^---------^
```

There is no guarantee that an object of type `T` *implements the* `Comparable<T>` *interface*

(c) Does the following declaration of `max3` work?

```
<T> T max3 (T a, Comparable<T> b, Comparable<T> c)
```

As `a`*,* `b` *and* `c` *are of type* `Comparable<T>`*, there is a type mismatch when assigning* `T max = a`*. Although an explicit type-casting can be used, e.g.* `T max = (T) a`*, type-casting is generally to be avoided.*

```
jshell> @SuppressWarnings("unchecked")
   ...> <T> T max3(T a, Comparable<T> b, Comparable<T> c) {
   ...>     T max = a;
   ...>     if (b.compareTo(max) > 0) {
   ...>         max = (T) b;
   ...>     }
   ...>     if (c.compareTo(max) > 0) {
   ...>         max = (T) c;
   ...>     }
```

```
   ...>     return max;
   ...> }
jshell> max3(-1, 2, -3)
$.. ==> 2
```

(d) To restrict `T` to have the `compareTo` method, i.e. any class that binds to `T` must implement `Comparable`, we redefine the type parameter `<T>` to be `<T extends Comparable<T>>`.

```
<T extends Comparable<T>> T max3(T a, T b, T c) {
    T max = a;
    if (b.compareTo(max) > 0) {
        max = b;
    }
    if (c.compareTo(max) > 0) {
        max = c;
    }
    return max;
}
```

Demonstrate how the method `max3` can be used to find the maximum of three values -1, 2 and -3. Explain how it works now.

According to the Java API Specification, the `Integer` *class implements* `Comparable<Integer>` *and hence the* `compareTo` *method is implemented.*

```
jshell> max3(-1, 2, -3)
$.. ==> 2
```

2. Suppose a `Fruit` class implements the `Comparable` interface, and `Orange` is a sub-class of `Fruit`.

```
class Fruit implements Comparable<Fruit> {
    @Override
    public int compareTo(Fruit f) { ... }
}

class Orange extends Fruit { }
```

We would like to redefine the `max3` method such that the parameter type of `max3` is `List<T>` instead (more specifically a list of three elements). Does the following declaration of the method work?

```
<T extends Comparable<T>> T max3(List<T> list)
```

Try it out by finding the maximum of a list of three fruits or a list of three oranges. How do you declare the method so that it works for both types of list? You should aim to make the method as flexible as you can.

Suppose we have:

```
class Fruit implements Comparable<Fruit> {
    @Override
    public int compareTo(Fruit f) { return 0; }
}
class Orange extends Fruit { }
```

Just declaring

`<T extends Comparable<T>> T max3(List<T> list)`

would work for `List<Fruit>` *only, but not for* `List<Orange>`*, since*

`Orange extends Comparable<Orange>` *does not hold.*

- *Alternative #1 — modify the generic type declaration:*

  ```
  jshell> <T extends Comparable<? super T>> T max3(List<T> list) {
     ...>     return list.get(0); // just return the first element for simplicity
     ...> }
  |  created method max3(List<T>)
  ```

  Now what can `T` *be bound to?* `T` *can certainly be bound to* `Fruit`

  ```
  jshell> List<Fruit> fruits = List.of(new Fruit(), new Orange())
  fruits ==> [Fruit@27973e9b, Orange@312b1dae]

  jshell> max3(fruits) // T in max3 bound to Fruit
  $.. ==> Fruit@27973e9b
  ```

  Not only that, notice that

  `Orange <: Fruit <: Comparable<Fruit> <: Comparable<? super Orange>`

  So `T` *can be bound to* `Orange`*!*

  ```
  jshell> List<Orange> oranges = List.of(new Orange(), new Orange())
  oranges ==> [Orange@27bc2616, Orange@3941a79c]

  jshell> max3(oranges) // T in max3 bound to Orange
  $.. ==> Orange@27bc2616
  ```

- *Alternative #2 — modify the method parameter:*

  ```
  jshell> <T extends Comparable<T>> T max3(List<? extends T> list) {
     ...>     return list.get(0);
     ...> }
  |  created method max3(List<? extends T>)

  jshell> max3(fruits) // T in max3 bound to Fruit
  $.. ==> Fruit@27973e9b
  ```

  What about `max3(oranges)`*? What can* `T` *be bound to?*

  - *Can it be* `Orange`*? Notice that* `<T extends Comparable<T>>` *would not work for* `List<Orange>`*, since* `Orange extends Comparable<Orange>` *does not hold.*

– *How about binding* `T` *to* `Fruit`*? Clearly,* `Fruit extends Comparable<Fruit>` *holds. And is* `List<Orange>` *a sub-type of* `List<? extends Fruit>`*? Yes!*

```
jshell> max3(oranges) // T in max3 bound to Fruit!
$.. ==> Orange@27bc2616
```

Notice that in this case, `Fruit` *is returned, not* `Orange`*!*

```
jshell> Orange o = max3(oranges)
|  Error:
|  incompatible types: inference variable T has incompatible bounds
|      equality constraints: Fruit
|      lower bounds: Orange,java.lang.Comparable<T>
|  Orange o = max3(oranges);
|             ^-----------^
```

Here's something worth pondering...

Suppose we have `Mandarin` *inheriting from* `Oranges`*:*

```
jshell> class Mandarin extends Orange {}
|  created class Mandarin

jshell> List<Mandarin> mandarins = List.of(new Mandarin())
mandarins ==> [Mandarin@30dae81]
```

How do we pass to `max3` *a* `List<Mandarin>` *and get back* `Orange`*? We should be able to do this since a mandarin is an orange. But using alternative #1 binds* `T` *to* `Mandarin`*; using alternative #2 binds* `T` *to* `Fruit`*!*

Interestingly for alternative #1,

```
jshell> <T extends Comparable<? super T>> T max3(List<T> list) {
   ...>     return list.get(0);
   ...> }
|  created method max3(List<T>)

jshell> Orange o = max3(mandarins) // T in max3 bound to Mandarin
o ==> Mandarin@30dae81
```

Although the above looks like `max3` *is returning* `Orange`*, but in actual fact,* `Mandarin` *is returned an assigned to* `Orange` *(which is valid since* `Mandarin <: Orange`*).*

We need a way to "force" the binding to the return type. Let's define `max3` *as a static method in a class* `X`*.*

```
jshell> class X {
   ...>    static <T extends Comparable<? super T>> T max3(List<T> list) {
   ...>       return list.get(0);
   ...>    }
   ...> }
|  created class X

jshell> X.<Orange>max3(mandarins)
|  Error:
|  incompatible types: java.util.List<Mandarin> cannot be converted to java.util.List<Orange>
|  X.<Orange>max3(mandarins)
|                 ^-------^
```

Notice from the above that T cannot be bound to `Orange`.

As such, `max3` *can be defined more generally by combining the two alternatives:*

```
jshell> class X {
   ...>    static <T extends Comparable<? super T>> T max3(List<? extends T> list) {
   ...>       return list.get(0);
   ...>    }
   ...> }
|  modified class X

jshell> X.<Orange>max3(mandarins) // T in X::max3 bound to Orange
$.. ==> Mandarin@30dae81
```

The following shows that the method call to `max3` *does indeed return an* `Orange`.

```
jshell> Mandarin m = X.<Orange>max3(mandarins) // Orange is not Mandarin!
|  Error:
|  incompatible types: Orange cannot be converted to Mandarin
|  Mandarin m = X.<Orange>max3(mandarins);
|               ^-----------------------^
```

And to exhaustively test out all possibilities:

```
jshell> X.<Fruit>max3(fruits)
$.. ==> Fruit@27973e9b

jshell> X.<Fruit>max3(oranges)
$.. ==> Orange@27bc2616

jshell> X.<Fruit>max3(mandarins)
$.. ==> Mandarin@30dae81

jshell> X.<Orange>max3(oranges)
$.. ==> Orange@27bc2616

jshell> X.<Orange>max3(mandarins)
$.. ==> Mandarin@30dae81

jshell> X.<Mandarin>max3(mandarins)
$.. ==> Mandarin@30dae81
```