CS2030 Programming Methodology
Semester 1 2019/2020

13 September 2019
Problem Set #3 Suggested Guidance
Generics and Variance of Types

1. For each of the statements below, indicate if it is a valid statement with no compilation error. Explain why.

   (a) `List<?> list = new ArrayList<String>();`

   (b) `List<? super Integer> list = new List<Object>();`

   (c) `List<? extends Object> list = new LinkedList<Object>();`

   (d) `List<? super Integer> list = new LinkedList<int>();`

   (e) `List<? super Integer> list = new LinkedList();`

   (a) Yes, since `ArrayList<String> <: List<String> <: List<?>`

   (b) No, List is an interface. It wil be fine if we change it to `ArrayList<Object>` *since* `ArrayList<Object> <: List<Object> <: List<? super Object> <: List<? super Integer>`

   (c) Yes, since `LinkedList<Object> <: LinkedList<? extends Object> <: List<? extends Object>`

   (d) Error. A generic type cannot be primitive type.

   (e) Compiles, but with a unchecked conversion warning. Use of raw type should also be generally be avoided.

2. Given the following Java program fragment,

```
class Main {
    public static void main(String[] args) {
        double sum = 0.0;

        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            sum += i;
        }
    }
}
```

   you can determine how long it takes to run the program using the `time` utility

```
$time java Main
```

   Now, replace `double` with the wrapper class `Double` instead. Determine how long it takes to run the program now. What inferences can you make?

   Despite it's conveniences, there is an associated overhead in the use of autoboxing. In addition, due to immutability of `Integer`*, many objects are created.*

3. Recall that the == operator compares only references, i.e. whether the two references are pointing to the same object. On the other hand, the `equals` method is more flexible in that it can override the method specified in the `Object` class.

   In particular, for the `Integer` class, the `equals` method has been overridden to compare if the corresponding `int` values are the same or otherwise.

   What do you think is the outcome of the following program fragment?

```
Integer x = 1;
Integer y = 1;
x == y

x = 1000;
y = 1000;
x == y
```

   Why do you think this happens? *Hint: check out Integer caching*

   We would expect the top fragment to be false since we are comparing object references. Since integers within a small range ar very often used, it makes sense for the `Integer` *class to keeps a cache of Integer objects within this range (-128 to 127) such that autoboxing, literals and uses of Integer.valueOf() will return instances from that cache instead.*

   Rather than concern oneself with the effects of caching or otherwise, the bottomline is to always use equals to compare two reference variables.

4. Compile and run the following program fragments and explain your observations.

   (a)
```
import java.util.List;

class A {
    void foo(List<Integer> integerList) {}
    void foo(List<String> StringList) {}
}
```

   (b)
```
class B<T> {
    T x;
    static T y;
}
```

   (c)
```
class C<T> {
    static int b = 0;

    C() {
        this.b++;
    }

    public static void main(String[] args) {
```

```
        C<Integer> x = new C<>();
        C<String> y = new C<>();

        System.out.println(x.b);
        System.out.println(y.b);
    }
}
```

(a) *Overloaded*

```
class A {
    void foo(List<Object> integerList) {}
    void foo(List<Object> StringList) {}
}
```

(b) *There is only one class* `B`. *For the field declaration* `T x`, *the type of* `X` *is bounded to the type argument* `T`, *this is fine for instance fields. However for class fields, there is only one copy of* `y`. *Which type argument should it be bounded to?*

(c) `2`
`2`
Although it seems there are two different classes, `C<Integer>` *and* `C<String>`, *there is still only one class* `C`. *There is only one copy of the class variable* `b`.

5. In the lecture, we have seen the generic method `max3` that takes in an array of generic type `T` that such that `T` implements the `Comparable` interface.

```
public static <T extends Comparable<T>> T max3(T[] arr) {
    T max = arr[0];
    if (arr[1].compareTo(max) > 0) {
        max = arr[1];
    }
    if (arr[2].compareTo(max) > 0) {
        max = arr[2];
    }
    return max;
}
```

What happens if we replace the method header with each of the following:

(a) `public static <T> Comparable<T> max3(Comparable<T>[] arr)`
If we declare `max` *with type* `Comparable<T>`, *then we require a cast*
`nums[1].compareTo((T)max)`
Also, realize that the method returns a `Comparable` *object.*

(b) `public static <T> T max3 (Comparable<T>[] arr)`
The above preserves the return type as `T`. *Suppose we declare* `max` *as type* `T` *now. Still, explicit casting is required when assigning an element of* `arr` *to* `max`, *e.g.*
`T max = (T) arr[0]`

(c) `public static Comparable max3(Comparable[] arr)`

This code fragment shows the effect of type erasure. When the compiler replaces the type-parameter information with the bound in the method declaration, it also inserts explicit cast operations in front of each method call to ensure that the returned value is of the type expected by the caller. Example,

```
(Integer) max3(new Integer[]{2, 3, 1})
```

What if the parameter type of `max3` is `List<T>` instead? How would you change the method header to be 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 `public static <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.*

The first solution is to modify the argument:

```
public static <T extends Comparable<T>> T max3(List<? extends T> list)
```

Now 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! This is a* **covariant** *relation.*

On the other hand, `Orange <: Comparable<Orange>` *does not hold since* `Orange <: Fruit <: Comparable<Fruit>`*, but* `Comparable<Fruit>` *and* `Comparable<Orange>` *are* **invariant**.

Another way is to declare it as

```
public static <T extends Comparable<? super T>> T max3(List<T> list)
```

Now what can `T` *be bound to? Notice that*

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

So `T` *can be bounded to* `Orange`*! Notice that the relation* `Comparator<Fruit> <: Comparator<? super Orange>` *is* **contravariant**.

And to be even more general, we should have:

```
public static <T extends Comparable<? super T>> T max3(List<? extends T>
list)
```

The use of the declaration `<T extends Comparable<? super T>>` *is very common all over Java's API. As such we can define* `max3` *as*

6. Which of the following code fragments will compile? If so, what is printed?

   (a)
   ```
   List<Integer> list = new ArrayList<>();
   int one = 1;
   Integer two = 2;

   list.add(one);
   list.add(two);
   list.add(3);

   for (Integer num : list) {
       System.out.println(num);
   }
   ```

   (b)
   ```
   List<Integer> list = new ArrayList<>();
   int one = 1;
   Integer two = 2;

   list.add(one);
   list.add(two);
   list.add(3);

   for (int num : list) {
       System.out.println(num);
   }
   ```

   (c)
   ```
   List<Integer> list = Arrays.asList(1, 2, 3);

   for (Double num : list) {
         System.out.println(num);
   }
   ```

   (d)
   ```
   List<Integer> list = Arrays.asList(1, 2, 3);

   for (double num : list) {
       System.out.println(num);
   }
   ```

(e)
```
List<Integer> list = new LinkedList<>();
list.add(5);
list.add(4);
list.add(3);
list.add(2);
list.add(1);

Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

(a)
```
1
2
3
```

(b)
```
1
2
3
```

(c)
```
prog.java:8: error: incompatible types: Integer cannot be converted to Double
for (Double num : list) {
                  ^
1 error
```

(d)
```
1.0
2.0
3.0
```

(e)
```
5
4
3
2
1
```