

Lab 11

1. Consider the following code fragments. For each, if there is a compiler error, identify where it occurs.

a. First fragment:

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
List<Number> nums = ints;
nums.add(3.14);
```

b. Second fragment:

```
List<Integer> ints = ArrayList<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(3.14);
```

2. A *group* is a collection of elements having one special element. An example of a group is the set of integers $\{ \dots -2, -1, 0, 1, 2, \dots \}$, with special element 0.

Here is a representation of a group as a Java class:

```
public class Group<T> {
    private T specialElement;
    private List<T> elements = new ArrayList<>();
    public Group(T special, List<T> elements) {
        this.specialElement = special;
        this.elements = elements;
    }
}
```

The following static method attempts to make a copy of a given instance of a Group, reproducing the state of the group in the copy.

```

public static Group<?> copy(Group<?> group) {
    List<?> elements = group.getElements();
    Group<?> grp = new Group<?>(group.getSpecialElement(), elements);
    return grp;
}

```

The code does not compile. Fix the code by capturing the wildcard with a helper method. Startup code is provided in the directory for this lab problem. Use the main method provided there to test your implementation. Note that the Group class has a toString method that will help in your test.

3. Draw a class diagram showing the inheritance relationships among the following types:

```

List<Integer>, List<Number>, List<? extends Integer>,
List<? extends Number>, List<? super Integer>, List<? super Number>,
List<?>, List<Object>

```

4. Recall the definition of sum given in the slides:

```

public static double sum(Collection<? extends Number> nums {
    double s = 0.0;
    for(Number num : nums) s += num.doubleValue();
    return s;
}

```

- a. Is there a compiler error in the following lines of code? If so, where?

```

List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
double dbl = sum(nums);
nums.add(3.14);

```

- b. Is there a compiler error in the following lines of code? If so, where?

```

List<Object> objs = new ArrayList<>();
objs.add(1);
objs.add("two");

```

```
List<? super Integer> ints = objs;
ints.add(3);
double dbl = sum(ints);
```

5. Create a generic programming solution to the problem of finding the second smallest element in a list. In other words, devise a `public static` method `secondSmallest` so that it can handle the biggest possible range of types.
6. Generalize the `contains` method for a `List` in the following way. First consider a simple implementation for a `List` of `Strings`:

```
public static boolean contains1(List<String> list, String s) {
    for(String x: list) {
        if(x == null && s == null) return true;
        if(s == null || x == null) continue;
        if(x.equals(s)) return true;
    }
    return false;
}
```

This `contains` method is tested in the following test method:

```
public static void test1() {
    List<String> list = Arrays.asList("Bob", "Joe", "Tom");
    boolean result = Main.contains1(list, "Tom");
    System.out.println(result);
}
```

We would like to generalize to a type variable `T`. Write the code for the most general possible `contains` method so that the type `T` can represent `Employees`, `Accounts`, `Managers`, and other types. Note that in more general lists, the objects in the `List` may not have overridden the `equals` method. Note also that classes like `Employee` and `Account`, which are provided in your startup code, do not have their own `equals` method, and you are not allowed to modify entity classes by overriding equals in them. To handle the lack of an `equals` method, you must find an alternative strategy.