OOP and Data Structures	Discussion 4 — Generics practice
CS2110 Summer 2019	Due 5PM 7/10

Before working on this discussion, you should read:

- java.util.Comparator.compare method specification

The `Comparator<T>` interface represents a method used to compare two objects. This way you can sort objects using different criteria (smallest to largest, largest to smallest, based on name, social security number, whatever).

This is an example of the *strategy design pattern*: a way to structure your code to make it more flexible and reusable.

In the provided class `Comparators`, I have created a simple type hierarchy for numbers. You should briefly peruse the first section of this file so that you understand the `Number`, `Integer`, and `Double` types. You will then implement some generic methods to work with comparators. Note that many of these types already exist in the standard library (and using those is generally preferred to rolling your own), but we will reimplement them today.

1. Implement the `NumberComparator` class, and check that `testNumberComparator` passes.
2. Implement the `IntegerComparator` class, and check that `testIntegerComparator` passes.
3. Class `ReverseComparator` stores a reference to another comparator and reverses it's order[1]. Complete the `ReverseComparator` class. Ensure that `testReverseComparator` passes.
4. An *iterator* is an object that represents a position in a list. The `Iterator<T>` type has two methods: `boolean hasNext()` and `T next()`. Each call to `next()` causes the iterator to return the element at the current position, and advances the position to the next in the list. Thus the elements of the list can be output by repeatedly calling `next()`:

```
Iterator<Integer> ints = Arrays.asList({5,4,1,2}).iterator();
ints.hasNext(); // returns true
ints.next(); // returns 5
ints.next(); // returns 4
ints.next(); // returns 1
ints.next(); // returns 2
ints.hasNext(); // returns false
ints.next(); // throws an exception
```

   The method `merge` takes in two input iterators and an output list. The input iterators will enumerate two sorted lists. The merge method should place all of the objects output by both iterators into the result list, in sorted order.

   Implement the `merge` method. Check that `testMerge` passes.

5. The type of `merge` is not as general as it could be. For example, it is impossible to merge a list of `Integers` and a list of `Doubles` into a list of `Numbers`. Change the argument types using `<? super E>` or `<? extends E>` as appropriate.
6. For additional practice with generic types (you don't need to submit this), download the `Animal` example from the lecture 8 notes. Experiment with the is-a relation by first making a hypothesis (e.g. I think `Collection<Dog>` is-a `Collection<?>`). Give a reason why you think this is the case (e.g. "`Collection<?>` just requires a `Object get()` method, and `Collection<Dog>`'s `Dog get()` method is even better"). Then, test it out in the `main` method: if a `Collection<Dog>` is a `Collection<?>` then you should be able to assign a `Collection<Dog>` object into a `Collection<?>` variable. You should be able to figure out the relationships between the following types: `Collection<Animal>`, `Collection<Dog>`, `Collection<? extends Animal>`, `Collection<? extends Dog>`, `Collection<? super Animal>`, `Collection<? super Dog>`, `Collection<?>`,

[1]Having one object wrap up another and modify or enhance it's behavior in some fashion is sometimes referred to as the *decorator pattern*.