CSCE 314 [Sections 500, 501] Programming Languages – Spring 2020
Hyunyoung Lee
Assignment 6
Assigned on Saturday, April is 4, 2020

Electronic submission to eCampus due **at 10:00 p.m., Wednesday, April 15, 2020**

By electronically submitting this assignment to eCampus by logging in to your account, you are signing electronically on the following Aggie Honor Code:

"On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment."

In this assignment, you will practice Java generics.

You will earn total 180 points.

Note 1: This homework set is **individual** homework, not a team-based effort. Discussion of the concept is encouraged, but actual write-up of the solutions must be done individually.

Note 2: Turn in one `yourLastName-yourFirstName-hw6.zip` file on eCampus, nothing else. Your zip directory must contain all your Java programs and a README file that explains how to compile and execute your code, and what is the expected output of your code when run. **The README file is worth ten points** (see below).

Note 3: All Java code that you submit must compile without any error using `javac` of Java 8 (or higher version). If your code does not compile, you will likely receive zero points for this assignment.

Note 4: Remember to put the head comment in your files, including your name, your UIN, and *acknowledgements of any help received* in doing this assignment. Again, **remember the honor code.**

Important: Before working on the problems in this assignment, carefully read Chapter 11 Generic Types of our textbook.

Problem 0. (10 Points) README. Explain in a README.txt file how to compile (using the javac command) and execute your codes (using the java command), and what is the expected output of your codes when tested. It should be detailed enough so that whoever grades your codes can understand what you were doing with your code clearly and should be able to reproduce your tests following what you wrote in it.

Problem 1. (70 Points) Implement generic interfaces. First study the `Cell` class that is given in pages 247–248. It can represent a generic singly linked list. You will modify the `Cell` class as below: call it `MyNode`:

```
public final class MyNode<E> { // class header needs to be modified also (see below)
  private E data;   // data field must be private
  private MyNode<E> next; // next field must also be private
  public MyNode (E val, MyNode<E> node) { . . . } //constructor
  // other necessary methods such as getter/setter and iterator() (see below)
}
```

Task 1. (30 points for the `MyNodeIterator<E>` class) As an **inner class** of `MyNode`, define class `MyNodeIterator<E>` to iterate over the values stored in a linked list of `MyNode<E>` objects. To do so, have your `MyNodeIterator<E>` class implement the `java.util.Iterator<E>` interface (see `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html`), with the class header `class MyNodeIterator<E> implements Iterator<E>`. Also, let it have a private field `MyNode<E> p`.

The constructor of class `MyNodeIterator<E>` should take a `MyNode<E>` as an argument, and thus have the header: `public MyNodeIterator (MyNode<E> n)`.

Task 2. (20 points for making `MyNode<E>` class iterable) with the class header `public final class MyNode<E> implements Iterable<E>`. See `https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html`. To do so, you need to define the `iterator()` method that returns `MyNodeIterator<E>` for this `MyNode<E>` object; `public MyNodeIterator<E> iterator() {...}`.

Task 3. (20 points for the `MyNodeTest` class – use the skeleton code provided.) Now, if `list` is of type `MyNode<E>`, you should be able to iterate over `list` using Java's "for each" `for`-loop:

```
for (E e : list) { /* do something with e */ }
```

Implement the class `MyNodeTest` that contains the following four static methods:

(a) (5 points) `int_sum()` that accepts a linked list of type `MyNode<Integer>` and sums up the values in each node in the linked list, and

(b) (10 points) `num_sum()` accepts a linked list of type `MyNode` with any element type that extends `Number`, and sums up the values in each node in the linked list into a double value (since double is the largest type within the Number type). Use a *bounded wildcard*.

(c) (5 points) `print()` accepts a linked list of type `MyNode` with any element type and prints out the values in each node in the linked list.

(d) The method `main()` (provided) tests your `MyNode<E>` and `MyNodeIterator<E>` classes. In the `main()`, an Integer list `intlist` of type `MyNode<Integer>` and a Double list `doublelist` of type `MyNode<Double>` are created by explicitly invoking the `MyNode` constructor recursively. And then, we invoke the two methods `print()` and `int_sum()` passing `intlist` as the argument. Also,

invoke `print()` and `num_sum()` with `doublelist` as the argument. Furthermore, invoke `num_sum()` with `intlist` as the argument, and notice the power of bounded wildcards! As seen in the main(), the only collection structure you are allowed to use in this problem is your own `MyNode` and nothing else, that is, you cannot use existing Collection provided by Java such as `ArrayList` or `LinkedList`. Have fun!

Problem 2. (100 Points) Implement a nicer linked list. You may notice in Problem 1 that it is rather inconvenient to build lists with `MyNode`. Implement another generic class `MyList<E>`, in terms of `MyNode`, that has a nicer interface. Class `MyList<E>` must have private fields `MyNode<E> n` and `int length`.
To do Tasks 1, 2 and 3, the class header of `MyList<E>` should be
`public class MyList<E> implements Iterable<E>, Cloneable, Comparable<MyList<E>>`.

Task 1. (5 points) Make `MyList<E>` *iterable* by implementing the `Iterable<E>` interface and giving an implementation of `iterator()`.

Task 2. (10 points) Make `MyList<E>` *clonable* by implementing the `Cloneable` interface and overriding the `Object.clone()` method. You must give your own *explicit implementation* for the `clone()` method using the for-each loop.

Task 3. (10 points) Make `MyList<E>` *comparable* by implementing the `Comparable<MyList<E>>` interface and overriding the `compareTo()` method. The comparison criterion is the length of the lists.

Task 4. (10+5 = 15 points) Also, override the `Object.equals()` and `Object.hashCode()` methods. The `equals()` criteria are (i) the length and (ii) the contents of the lists (but not the order of the values in the list). For example, if `l1 = [1,2,3]`, `l2 = [2,1,3]`, and `l3 = [1,1,2,3]`, then `l1.equals(l2)` and `l2.equals(l1)` must return `true` but not `l1.equals(l3)` or `l3.equals(l1)` (nor for `l2` and `l3`).

[Hint: Since you will first check whether the lengths of the two lists are the same, your hashCode() can simply return the length of the list. If the lengths are the same, then check if the elements are all the same using a nested for-each loop.]

Task 5. (5 + 10 = 15 points) Define the two constructors for the `MyList<E>` class:

```
public MyList();  // create an empty list
public MyList(Iterable<E> iterable);
```

The second constructor should copy the elements in `iterable` to `MyNode`s of this list.

Task 6. (45 points) Implement these member methods (with their expected meaning, see below):

`public MyList<E> reverse();` (10 points)
`public String toString();` (10 points)
`public void push(E item);` (10 points)
`public E pop();` (10 points)
`public E peek();` (5 points)

Also include `public int getLength() { return length; };` (no points)

A call `x.reverse()` should reverse `x`, and return the reversed list as the result. An easy way to implement `reverse()` is to reconstruct a new list, and swap that in place of the original. The `toString()` method should print out the list in the following form: a list of integers 1, 2, and 3 should be printed as
`[(head: 1) -> (2) -> (3)]`.
The `push` method works the same way as the cons (`:`) operator in Haskell, i.e., it prepends `item` at the head of the list as the left-most element. The `pop` method removes the head (left-most) element from the list and returns its value. The `push` and `pop` method should modify the length of the list appropriately. The `peek` method returns the head element's value without removing the element.

Class `MyListTest` with a `main()` that tests those requirements is provided. Feel free to expand it to test your implementations. Include the `MyListTest` class in your .zip file.

Have fun!