CSCE 314 [Sections 595, 596, 597] Programming Languages – Spring 2024
Hyunyoung Lee
Homework Assignment 5
Assigned on Sunday, March 24, 2024

Electronic submission on Canvas due **at 11:59 p.m., Friday, April 5, 2024**

By electronically submitting this assignment to Canvas 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 inheritance, dynamic dispatching, and Java generics. You will earn total 135 points. Here are some general instructions.

1. This homework set is an *individual* homework, not a team-based effort. Discussion of the concept is encouraged, but actual write-up of the solutions must be done individually by yourself. Your final product – the code as well as comments, explanations, the README.txt file – should never be shared.

2. Read the problem descriptions and requirements carefully! There may be significant penalties for not fulfilling the requirements.

3. **Explain each line or block of your code in your own words in the comments.** Even though you think the code is self-explanatory, explain in your own words anyway! Your work will be graded not only on the correctness of your code, but also on the consistency and clarity with which you express it.

4. Turn in one *yourFirstName-yourLastName-hw5.zip* file on Canvas, nothing else. Your zip directory must include the two directories: P2 (for Problem 2 Shapes) and P3 (for the Cell linked list in Problems 3 and 4) and one **README.txt** file, and nothing else. Each of the two directories P2 and P3 must contain all of the corresponding .java files (and the input.txt file for P2) and *no .class files.*

   The **README.txt** file explains how to compile and execute your code, and what is the expected output of your code when tested. **The README.txt file is worth ten points.**[1] See below for more specific information.

5. All Java code that you submit must compile without errors using `javac` (at the terminal) of Java version 11 or higher. If your code does not compile using javac, you risk receiving zero points for the entire corresponding problem.

6. Remember to put the head comment in *all* of your files. In the skeleton code provided, fill in your name, your UIN, and and *acknowledgements of any help received* in doing this assignment. Do not remove anything from the head comment. Again, **remember the honor code!**

[1]Without README.txt, you risk getting points off if it is not clear (to whoever grades your code) how some functionalities are to be tested or how they are presented in the output. Thus, the README.txt file is actually a lot more worth than ten points!

Problem 1. (10 points) First, put your name and UIN at the top of the README.txt file (a plain .txt file). And then, in the README.txt file explain how to compile (using the javac command) and execute your code (using the java command), and what is the expected output of your code when tested (for each task). Each problem (and each task) should be clearly marked. It should be detailed enough so that whoever grades your code can understand what you were doing with your code clearly and should be able to reproduce your tests following what you wrote in a README.txt file. Here is a basic framework of your README.txt file:

```
Title: Homework 5 README
Name:
UIN:

----------
Problem 2
----------

(explain how to compile (using javac at the terminal) and execute your code
(using java at the terminal), and what is the expected output of your code
when tested (for each task/functionality))

----------
Problem 3
----------

(ditto)

----------
Problem 4
-----------

(ditto)
```

Problem 2. (40 points) Inheritance and dynamic dispatching using Shapes. Skeleton codes are provided.

A `Shape` class represents a geometric figure in some coordinate position. `Shape` allows for finding out its position and area with the methods `Point position()` and `double area()`. `Point` is a class that can represent a two-dimensional coordinate.
The `Shape` class must be an abstract class, from which you will derive two subclasses: `Square` and `Circle` to represent two different kinds of shapes. A square is defined by its upper-left corner (position), and the length of one side. A circle is defined by the center point (position) and the radius.

Tasks

1. Implement class `Point`.

2. Implement class `Shape`. Store one point in the class `Shape`, as each shape will need one point that serves as the position of the shape.
3. Implement the two subclasses: `Square` and `Circle`. Each of the two classes should inherit from the class `Shape` and define `area()` appropriately.
4. Add the `toString` method to each of the `Square` and `Circle` classes.
5. Implement a class `TotalAreaCalculator` with one static method `calculate(Shape[] shapes)` that will calculate the total area of an array of shapes.
6. The `main()` method is provided in class `Main`. Study the `main()` method carefully and make your class definitions above work with the provided `main()`. Do not modify `main()` except adding the parts that output the sorted shapes and the total area (see Task 10 below).

Defining equality. We define two different shapes to be equal if and only if (i) they are of the same kind (that is, `Circle` or `Square`), (ii) their position is the same, and (iii) the geometric figures they represent are equal (i.e., the two shapes are congruent). Textbook Section 3.8 discusses implementing equality.

Tasks

7. Implement the `equals` method for `Shape` and the two derived classes.
8. Override `hashCode` for these classes (as you should whenever you override `Object`'s `equals` method).

Comparison. Two shapes can be compared based on area, so that shape `a` is less than or equal to shape `b` if and only if `a`'s area is less than or equal to `b`'s area.

Tasks

9. Make this ordering the *natural ordering* of `Shape`s. (Sections 4.1 and 21.3 discuss natural orderings and making classes comparable).
10. Give the implementation toward the end of `Main.main` so that it also prints out the shapes in an increasing order according to your natural ordering, and calculates and outputs the total area of all of the given shapes.

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

Problem 3. (45 Points) Implement generic interfaces.

First study the `Cell` class that is given in the textbook pages 247–248 (in the beginning of Chapter 11). It can represent a generic singly linked list. You will modify the `Cell` class as below:

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

Task 1. (20 points for the `CellIterator<E>` class) Define class `CellIterator<E>` to iterate over the elements stored in a linked list of `Cell<E>`. To do so, have your `CellIterator<E>` class implement the `java.util.Iterator<E>` interface (see `https://docs.oracle.com/javase/10/docs/api/java/util/Iterator.html`), with the class header `class CellIterator<E> implements Iterator<E>`. Also, let it have a private field `Cell<E> p`.

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

Task 2. (10 points for making `Cell<E>` class iterable) This will be done with the class header `public final class Cell<E> implements Iterable<E>`. See `https://docs.oracle.com/javase/10/docs/api/java/lang/Iterable.html`. To do so, you need to define the `iterator()` method that returns `CellIterator<E>` for this `Cell<E>` object; `public CellIterator<E> iterator() {...}`.

Also, implement the constructor (with the header `public Cell (E elem, Cell<E> next)`) and the getter/setter methods.

Task 3. (15 points for the three static methods in the `CellTest` class – use the skeleton code provided.) Now, if `list` is of type `Cell<E>`, you should be able to iterate over `list` using Java's "for each" `for`-loop:

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

(a) (5 points) `int_sum()` accepts a linked list of type `Cell<Integer>` as an argument and sums up all of the element values in the linked list.

(b) (5 points) `num_sum()` accepts a linked list of type `Cell` of any element type that extends `Number` as its argument (use a *bounded wildcard*), and sums up the values in the linked list into a `double` value.

(c) (5 points) `print()` accepts a linked list of type `Cell` with any element type as its argument and prints out the element values in the linked list.

(d) The method `main()` (provided) tests your `Cell<E>` and `CellIterator<E>` classes. In the `main()`, an Integer list `intlist` of type `Cell<Integer>` and a Double list `doublelist` of type `Cell<Double>` are created by explicitly invoking the `Cell` 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 `Cell` and nothing else, that is, you are *not allowed to* use existing Collection provided by Java such as `ArrayList` or `LinkedList`.

An example output of CellTest.java is shown below.

```
===
1 22 21 12 24 17
sum of intlist is 97
sum of null list is 0
===
```

```
===
1.0 16.0 13.72 5.0 22.0 7.1
sum ints = 97.0
sum doubles = 64.82
===
```

Problem 4. (40 Points) Implement a nicer linked list.

You may notice in Problem 3 that it is rather inconvenient to build lists with `Cell`. Implement another generic class `CellList<E>`, in terms of `Cell`, that has a nicer interface. The class header of `CellList<E>` should be
`public class CellList<E> implements Iterable<E>, Cloneable, Comparable<CellList<E>>`.
Class `CellList<E>` must have private fields `Cell<E> n` and `int length`. The `iterator()` for `CellList<E>` simply returns the `iterator()` of `Cell<E>` (as given in the skeleton code). To implement the `Comparable<CellList<E>>` interface, we need to override the `compareTo()` method. The comparison criterion is the length of the lists. Its implementation is provided.

Task 1. (5 points) To implement the `Cloneable` interface, override the `Object.clone()` method. You must give your own *explicit implementation* for the `clone()` method using the for-each loop.

Task 2. (10 points) Also, override the `Object.equals()` method (`Object.hashCode()` is provided). 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`).

[Hint1: 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 (as given). If the lengths are the same, then check if the elements are all the same using a nested for-each loop.]

[Hint2: To check the equality of two lists where either one or both are with possibly duplicated elements, sort the two lists before checking element-by-element equality. You can use Arrays.sort(), for which you need to import java.util.Arrays.]

Task 3. (5 points) Define the one-arg constructor for the `CellList<E>` class:

```
public CellList(Iterable<E> iterable);
```

The no-arg constructor (`public CellList();`) that creates an empty list is provided. The one-arg constructor should copy the elements in `iterable` to `Cells` in this list. The element order in `iterable` must be preserved in the constructed list, that is,

```
CellList<Integer> list = new CellList<Integer>(Arrays.asList(1,2,3,4));
```

should construct `list` with `1` as its first (head) element, `2` its second element, `3` its third element, and `4` its last element. After you implement `toString()` (see below), `System.out.println("list = " + list);` should output

```
list = [(head: 1) -> (2) -> (3) -> (4)]
```

Task 4. (20 points) Implement the three methods – `toString()`, `push()`, and `pop()` with their expected meaning, see below. Two methods – `peek()` and `getLength()` – are provided:

`public String toString();` (8 points)
`public void push(E item);` (5 points)
`public E pop();` (7 points)

`public E peek() { return n.getVal(); }`
`public int getLength() { return length; }`

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` methods should modify the length of the list appropriately. The `peek` method returns the head element's value without removing the element.

Class `CellListTest` with a `main()` that tests those requirements is provided. Feel free to expand it to test your implementations. Don't forget to include the CellListTest.java file in your zip directory.

Have fun!

An example output of CellListTest.java:

```
stringlist = [(head: A) -> (the) -> (the) -> (dove)]
stringlist2 = [(head: A) -> (dove) -> (the) -> (the)]
stringlist3 = [(head: A) -> (dove) -> (dove) -> (the)]
stringlist equals to stringlist2 ? true
stringlist equals to stringlist3 ? false
CellList<Integer> equals to CellList<String> ? false
list  = [(head: 1) -> (2) -> (3) -> (4)]
list1 = [(head: 2) -> (4) -> (3) -> (1)]
list == list1 is false
list.equals(list1) = true
list3 = [(head: 1) -> (2) -> (3) -> (1)]
list4 = [(head: 1) -> (2) -> (3) -> (1) -> (4)]
list1.equals(list3) = false
list1.equals(list4) = false
list.compareTo(list1) = 0
list.compareTo(list4) = -1
[(head: )]
[(head: 1) -> (2) -> (3) -> (4)]
```

```
1
[(head: 22) -> (21) -> (2) -> (3) -> (4)]
22
[(head: 22) -> (21) -> (2) -> (3) -> (4)]
22 22
21 21
2 2
3 3
4 4
[(head: )]
list1 = [(head: 2) -> (4) -> (3) -> (1)]
list2 = [(head: 4) -> (3) -> (2) -> (21) -> (22) -> (1) -> (2) -> (3) -> (4)]
list2.compareTo(list1) = 1
=== end of test
```