Homework 08 – Waiting in Line…

Authors: Lucas, Eric, Daniel, Sang Yoon
Topics: Generics, ADTs, Iterator, Iterable

Problem Description

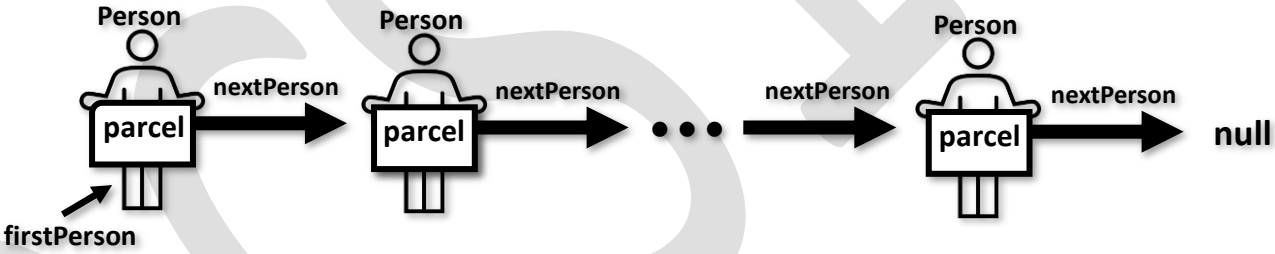
As a software engineer for one of the world's biggest supermarket chains, FooMart, you are responsible for writing code to organize the line of people who are not satisfied with some products they purchased and would like to return it. You opt for what you feel is an elegant solution: a data structure that represents a line of people carrying products they are unhappy with in neatly wrapped parcels.

Solution Description

We highly recommend that you **read this document in its entirety** and familiarize yourself with the requirements of this assignment **before writing any code**!

Important: All classes created in this assignment should be **generic classes**. All instances of generic classes should use generic types and should never use raw types or `Object`. If a method returns a reference type, it must use generic types, and should never use raw types or `Object`.

A `Line` is an ordered sequence of `Person` instances. Each person carries a `parcel` and a reference to the `nextPerson` in line. A reference to the first person in line should be kept in `firstPerson`. When the line is empty, we represent that there is not a first by setting the value of `firstPerson` to `null`.



We have provided you with a file named List.java. This file is an interface named `List` that you will implement in Line.java. It represents the List ADT and extends from the `Iterable` interface. You will need to complete and turn in 3 classes: Person.java, LineIterator.java, and Line.java.

Notes:

1. All variables should be inaccessible from other classes and require an instance to be accessed through, unless specified otherwise.
2. All methods should be accessible from everywhere and must require an instance to be accessed through, unless specified otherwise.
3. Use constructor chaining and reuse methods whenever possible! Ensure that your constructor chaining helps you maximize code reuse!
4. Reuse code when possible and helper methods are optional.
5. Make sure to add Javadoc comments to your methods and classes!
6. All exceptions thrown must have a descriptive message.

7. There are a lot of edge cases to consider when completing this assignment. Make sure to perform extensive testing in a Driver file! We encourage sharing your tests and outputs in the HW08 Discussion Thread on Ed Discussion.

List.java

This generic interface defines the List ADT and contains the methods that the `Line` class must implement. The Javadocs in this file will contain more information about each of the methods you must implement including what the method does, what the parameters represent, what should be returned, and when to throw exceptions. **Read the Javadocs carefully.**

This interface extends from the `Iterable` interface, so the `iterator` method must also be implemented. Refer to the Java documentation on **Iterable** for more information about this interface.

Methods:

In addition to any methods required by the `Iterable` interface, the `List` interface requires the following methods to be implemented:

Note: You do not need to implement the `forEach` and `spliterator` methods of `Iterable`.

- `void add(E element)`
- `void add(int index, E element)`
- `E remove()`
- `E remove(int index)`
- `E remove(E element)`
- `E set(int index, E element)`
- `E get(int index)`
- `boolean contains(E element)`
- `void clear()`
- `boolean isEmpty()`
- `int size()`

Person.java

This generic class represents a person in line. A person should be able to carry a parcel of any type.

Note: You may NOT create any instance variables nor instance methods that are not required.

Generic Type:

- `T` – the type of parcel this person carries

Variables:

- `parcel` – the parcel this person carries
  - If we ever attempt to set its value to null, throw an `IllegalArgumentException`.
- `nextPerson` – a reference to the next person in line, which should also carry the same type of parcel as *this* person

Constructors:

- A constructor that takes in `parcel` and `nextPerson`.
- A constructor that takes in `parcel` and defaults `nextPerson` to null.

Methods:

- Getters and setters for each of the variables.

LineIterator.java

This is a generic class that represents an iterator to iterate over a `Line`. It implements the `Iterator` interface using generics. Refer to the Java documentation on **Iterator** for more information.

For `Line` to successfully implement the `Iterable` interface (which the provided `List` interface extends from), you must provide an implementation for the `iterator` method. The `iterator` method returns an instance of an `Iterator` that knows how to iterate over the Iterable.

To successfully implement the `iterator` method, we will create our own iterator, `LineIterator`, which has knowledge about how to iterate over a `Line`. A `LineIterator` should iterate over every Person in a Line, in the same order that each person is ordered in the line.

Note: You may NOT create any instance variables nor instance methods that are not required.

Generic Type:

- `T` – the type of parcel that persons in the line carry

Variables:

- `nextPerson` – the next person in line to iterate over
  - **Hint*:*** When a `LineIterator` is first instantiated, `nextPerson` should store a reference to the first person in line.

Constructor:

- A constructor that takes in a `Line` that *this* instance of `LineIterator` iterates over.
  - If the `Line` passed in is null, throw an `IllegalArgumentException`.

Methods:

- `hasNext()`
  - This method properly overrides the `hasNext` method required by `Iterator`.
  - Returns a boolean specifying whether there is another person to iterate over.
    - **Hint*:*** What value of `nextPerson` indicates the existence of a person?
- `next()`
  - This method properly overrides the `next` method required by `Iterator`.
  - Returns the `parcel` held by the next person in line and advances the iterator to the person that is after them in line.
    - **Note: The return type of this method should not be `Person`, but instead the type of the `parcel` each person carries.**
  - **Hint:** The first time `next` is called, it should return the parcel held by the first person in line and advance the iterator to the second person. The second time `next` is called, it should return the parcel held by the second person in line and advance the iterator to the third person in line, and so on.
  - If there are no more parcels to iterate over and `next` is called, throw a `NoSuchElementException`.

Functionality:

- After properly implementing `LineIterator`, you should be able to use the iterator to print out all the `parcels` held by persons in a `Line` using the following block of code. Assume that `line` is an instance of `Line`.

```
Iterator<T> itr = line.iterator();
while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

Line.java

This generic class represents a line. A `Line` is comprised of `Person` instances that are linked together. This class implements all the methods required by the provided `List` interface (don't forget about the methods required by the `Iterable` interface). It utilizes the `Person` and `LineIterator` classes to satisfy the requirements.

Note: Line.java is a provided file with fields and method stubs. You just need to provide the methods' implementation. The `toString` method has been implemented to help you visualize the Line.

Note: You may NOT create any instance variables that are not required. Private helpers are allowed.

Generic Type:

- `T` – the type of parcel that persons in this line carry

Variables:

- `firstPerson` – a reference to the first person in line
  - **Hint:** Think about which part of a linked list this refers to.
- `size` – the number of people in line
  - Don't forget to update this value when a person joins or leaves the line!

Constructors:

- A constructor that takes in an array of parcels of type `T`.
  - The parcels in this `Line` must be in the same order as the parcels in the passed in array.
  - If the passed in array is null or contains any null elements, throw an `IllegalArgumentException`.
  - Don't forget to correctly update the value of `size`.
- A no-argument constructor that initializes an empty `Line`.

Methods:

***In addition to** the methods required by the List interface*, implement the following methods:

Note: There are many edge cases to consider when implementing List, so be sure to account for them!

- `toArray()`
  - Return the parcel that each `Person` carries in an array.
  - You must implement this method by explicitly using an iterator.
  - The order of the parcels in the array and the `Line` should be the same.
  - **Note: The intuitive approach of creating a generic array in Java by writing `T[] arr = new T[size];` causes a compiling error. Instead, create an Object array and cast that to a generic array:** `T[] arr = (T[]) new Object[size];`
    - This cast is not checked by the compiler, so it will display a warning (not an error) to notify you of an unchecked or unsafe operation. It is acceptable if the

only warnings you receive are related to performing an unchecked cast to a generic array.
    - You can recompile with the `-Xlint` flag before the file name to see more specific details about the warnings.

- `reverse()`
    - **Recursively** reverse the order of the `Person` instances in the `Line` such that the person who is first in line becomes the last in line, the person who is second in line becomes the second to last in line, and so on.
    - This method should not accept any parameters and does not return anything.
    - This line must be reversed in-place, meaning auxiliary data structures should not be created. You should be modifying the linked list through `Line`'s `firstPerson` reference.
    - Once you have reversed the line, make sure to update the `firstPerson` reference of the `Line` so that it points to the new first person in the line.
    - You will likely need to write a recursive helper method.
    - **Hint:** The basis of recursion is to solve a big problem by solving smaller subproblems.
        - Remember that a recursive solution must meet three requirements.
            - Presence of **at least one base case**.
            - Make **progress towards the base case**. Define the problem in terms of a smaller subproblem. We'll use [A → B → C → null] as an example line.
                - View each person as the "firstPerson" of a smaller line of persons following them. That is,
                    - A is the "firstPerson" of the original line,
                    - B is the "firstPerson" of line [B → C → null],
                    - C is the "firstPerson" and only person of line [C → null],
                    - and null is the "firstPerson" of an empty line.
                - You can view the smaller subproblem as reversing a smaller line.
                - To reverse the order of the line, each person needs to become the nextPerson of their own nextPerson. That is,
                    - A should become the nextPerson of B,
                    - B should become the nextPerson of C,
                    - and null should become the nextPerson of A.
            - Make a **recursive call**.
                - Call the same method to solve a smaller problem. How can you update the parameter(s) to progress towards the base case?
        - Considering these points, think about what parameter(s) may be helpful to implement this method recursively.
        - We recommend that you solve this problem by diagramming before translating your solution into code. You also don't have to follow our suggestions as there is more than one solution!

Checkstyle

You must run Checkstyle on your submission (to learn more about Checkstyle, check out cs1331-style-guide.pdf under the Checkstyle Resources module on Canvas). **The Checkstyle cap for this assignment is 50 points.** This means there is a maximum point deduction of 50. If you don't have Checkstyle yet,

download it from Canvas → Modules → Checkstyle Resources → checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar YourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the number of points we would take off (limited by the Checkstyle cap). In future assignments we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must Javadoc your code.

Run the following to only check your Javadocs:

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both Javadocs and Checkstyle:

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

For additional help with Checkstyle see the CS 1331 Style Guide.

Turn-In Procedure

Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- `Person.java`
- `LineIterator.java`
- `Line.java`

Make sure you see the message stating the assignment was submitted successfully. From this point, Gradescope will run a basic autograder on your submission as discussed in the next section. **Any autograder tests are provided as a courtesy to help "sanity check" your work and you may not see all the test cases used to grade your work.** You are responsible for thoroughly testing your submission on your own to ensure you have fulfilled the requirements of this assignment. If you have questions about the requirements given, reach out to a TA or Professor via the class forum for clarification.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the assignment. We will only grade your latest submission. **Be sure to submit every file each time you resubmit**.

Gradescope Autograder

If an autograder is enabled for this assignment, you may be able to see the results of a few basic test cases on your code. Typically, tests will correspond to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue. **We reserve the right to hide any or all test cases, so you should make sure to test your code thoroughly against the assignment's requirements.**

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g., non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or Checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

Burden of Testing

You are responsible for thoroughly testing your submission against the written requirements to ensure you have fulfilled the requirements of this assignment.

Be **very careful** to note the way in which text output is formatted and spelled. Minor discrepancies could result in failed autograder cases.

If you have questions about the requirements given, reach out to a TA or Professor via the class forum for clarification.

Allowed Imports

To prevent trivialization of the assignment, you are only allowed import of the following classes:

- `java.util.Iterator`
- `java.util.NoSuchElementException`

Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our autograder. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`
- `System.arraycopy`

Collaboration

Only discussion of the assignment at a conceptual high level is allowed. You can discuss course concepts and assignments broadly; that is, at a conceptual level to increase your understanding. If you find yourself dropping to a level where specific Java code is being discussed, that is going too far. Those discussions should be reserved for the instructor and TAs. To be clear, you should never exchange code related to an assignment with anyone other than the instructor and TAs.

The only code you may share are test cases written in a Driver class. If you choose to share your Driver class, they should be posted to the assignment discussion thread on the course discussion forum. We encourage you to write test cases and share them with your classmates, but we will not verify their correctness (i.e., use them at your own risk).

Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items.
- Do not submit `.class` files.
- Test your code in addition to the basic checks on Gradescope.
- Submit every file each time you resubmit.
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points.
- **Check on Ed Discussion for a note containing all official clarifications and sample outputs.**

It is expected that everyone will follow the Student-Faculty Expectations document and the Student Code of Conduct. The professor expects a **positive, respectful, and engaged academic environment** inside the classroom, outside the classroom, in all electronic communications, on all file submissions, and on any document submitted throughout the duration of the course. **No inappropriate language is to be used, and any assignment deemed by the professor to contain inappropriate offensive language or threats will get a zero.** You are to use professionalism in your work. Violations of this conduct policy will be turned over to the Office of Student Integrity for misconduct.