Indiana University
CSCI-C212: Introduction to Software Systems (F24)
Joshua Crotts

Problem Set 5
November 6, 2024

Interfaces, Inheritance, Exceptions, File I/O

Important Dates:

- Assigned: November 6, 2024
- Deadline: December 4, 2024 at 11:59 PM EST

Objectives:

- Students become familiar with inheritance through lazy lists.
- Students understand the hierarchy imposed by interfaces and how they relate to storing instances of different subclasses in a collection.
- Students employ polymorphic method design to solve a problem.
- Students design a real-world data structure example through arbitrarily large natural numbers.
- Students work with simple file I/O and exception handling to parse strings and numbers.

What To Do:

Design classes with the given specification in each problem, along with the appropriate test suite. **Do not round your solutions!**

You must write sufficient tests and adequate documentation.

All problems, except for the extra credit problems, are listed in *Learning Java - A Test-Driven Approach*. This problem set contains eight required problems and three extra credit problems, meaning the maximum possible score on this problem set is 130%/100%. Extra credit problems are not eligible for corrections.

1. Exercise 4.31 [*Fibonacci lazy list*]
2. Exercise 4.32 [*Taking elements from a lazy list*]
3. Exercise 4.33 [*Passing functions to a lazy list*]
4. Exercise 4.34 [*Cyclic lazy list*]
5. Exercise 4.57 [*Arbitrarily large integers*] *(Note about this exercise: it is long, involved, and accounts for 25% of your A5 grade, so do not wait to start it. The autograder is extremely thorough when checking your solution. We fuzz-test your solution with thousands of inputs that are hundreds of digits long. We provide a few initial, separate tests, but the bulk of your points from the autograder come from the fuzz (randomized) tests. Be aware that your submission could fail because it is incorrect OR because it is too slow!)*

   (a) **Extra Credit (10%):** Division is intentionally omitted from the problem because the remaining operations are time-consuming to implement. If you have the time and want the extra points, you can implement a division algorithm for your `BigInt` class. You must provide a `BigInt div(BigInt divisor)` method to support dividing positive and negative arbitrarily large integers, and it must pass the autograder to receive credit. There is no partial credit. Some notes:

   - Divisions by zero should result in an `IllegalArgumentException` being thrown by the method.
   - You, of course, do not need to account for cases where the divisor is larger than the dividend.
   - The division algorithm should "round" all quotients *towards* zero. That is, $13/4$ should produce 3 and not 4. Similarly, $-17/4$ should produce -4 and not -5.

6. Exercise 5.3 [*Capitializing sentences*]
7. Exercise 5.4 [*Spellchecker*]
8. Exercise 5.15 [*Maze solver*]

9. **Extra Credit (10%):** *(This extra credit looks like a lot of work, but it really isn't! Our solution, without comments, is 37 lines long. Most of what follows is background information on Python for those who are new to the language.)* The Python programming language supports the use of `range(n)`, which returns a generator/stream/lazy list of integers in the interval $[0, n)$. For example, below is a segment of Python code that sums the numbers from 0 to a given value of n:

```
def sumNums(n):
  s = 0
  for i in range(n + 1):
    s += i
  return s

assert sumNums(5) == 15
```

It also supports a more powerful `range(start, end, step)` generator function, which returns a generator of integers in the interval `[start, end)` in changes of `step`.

```
def sumEvenNums(a, b):
  s = 0
  for i in range(a, b + 1, 2)
    s += i
  return s
```

In this exercise you will design the `RangeLazyList` class, which implements *three* interfaces: `ILazyList<Integer>`, `Iterator<Integer>`, and `Iterable<Integer>`. We know what two former interfaces do, but the third is one that we have not previously mentioned. Recall that several classes from the Collections framework, e.g., `ArrayList`, can be traversed over using the enhanced-for loop:

```
ArrayList<Integer> L = ...;
for (int x : L) {
  // Do something with x.
}
```

Well, if you have ever wondered *how* it's possible to make a class usable with the enhanced-for loop, this exercise will demonstrate. The enhanced-for loop described above is actually syntactic sugar for

```
for (Iterator<Integer> x : L.iterator()) {
  // ...
}
```

The enhanced-for loop simply abstracts the need to explicitly call the `iterator` of the type. Any class that can be traversed using the enhanced-for loop "via syntactic sugar" must implement `Iterable`.

(a) Design the `RangeLazyList` class, which implements all of the aforesaid interfaces. It should contain two private constructors:

- `private RangeLazyList(int s, int e, Function<Integer, Integer> f)`, which receives a starting integer `s`, an ending value `e`, and a unary function over integers `f`. The provided function determines how to go from the current number to the next. (This should be eerily similar to your `FunctionalLazyList` class!)
- `private RangeLazyList(int n)`, which receives an ending value `n`. Invoking this constructor should call the other constructor with `0, n, x -> x + 1`. In other words, calling the constructor with only one value produces a generator of integers from $[0, n)$ in one-step increments.

Of course, the class should store the necessary and relevant instance variables.

(b) Because your class implements `ILazyList<Integer>`, you must override `public Integer next()`. Override the method to return the next integer in the sequence. Again, the implementation should be nearly identical to the one used in `FunctionalLazyList`. One thing that you should remember is that implementing the `Iterator<Integer>` interface also means that you have to override the `public Integer next()` method, so we get two for the price of one.

(c) Because your class implements `Iterable<Integer>`, you must override `public Iterator<Integer> iterator()`. Of course, the `RangeLazyList` is definitionally an `Iterator<Integer>`, so just return `this`.

(d) Design the following static methods (if you're wondering why we privatize the constructors and instead opt to use static methods, it's because we want to simulate Python's `range` function):

- `static RangeLazyList range(int s, int e, Function<Integer, Integer> f)`, which returns an instance of a new `RangeLazyList` containing the provided fields.
- `static RangeLazyList range(int n)`, which returns an instance of a new `RangeLazyList` containing the provided field.

10. **Extra Credit (10%):** Similar to `range`, Python also has the `enumerate` function that, when given a traversable data structure, returns a generator where its items are paired with the index of that item. For example:

```
ls = ["Nebraska", "Butterscotch", "Blackie", "Bella"]
for item in enumerate(ls):
  print(item)

# Code outputs:
(0, "Nebraska")
(1, "Butterscotch")
(2, "Blackie")
(3, "Bella")
```

We can somewhat emulate this behavior in Java with our lazy list implementation, and you will do so in this exercise.

(a) Design the generic `EnumerateLazyList<T>` class that implements the following interfaces:

- `ILazyList<EnumerateLazyList.EnumerateItem<T>>`
- `Iterator<EnumerateLazyList.EnumerateItem<T>>`
- `Iterable<EnumerateLazyList.EnumerateItem<T>>`

This, of course, raises the question of what is `EnumerateLazyList.EnumerateItem<T>`. Java does not have native support for *tuples*, which is what the `enumerate` function in Python returns. So, our `EnumerateLazyList` class will contain a (non-private) static class called `EnumerateItem<T>`, which stores its index and the item at that index.

(b) Design the generic `EnumerateItem<T>` class as described. It should contain a relevant constructor and the accessors `int getIndex()` and `T getItem()`, which return the stored index and item respectively. Also, override its `public String toString()` method to return a stringified representation of the form `"(index, item)"`.

(c) Override the `public EnumerateItem<T> next()` method to return a new instance of `EnumerateItem` with the current index and the value at that index. Also be sure to update the stored index.

(d) Override the `public Iterator<EnumerateLazyList.EnumerateItem<T>> iterator()` method to return an instance of `this`.

(e) Design the `private EnumerateLazyList(List<T> ls)` constructor that stores the list to enumerate as an instance variable, as well as the index of the current item accessed by the iterator.

(f) Design the `static <T> EnumerateLazyList<T> enumerate(List<T> ls)` method that returns an enumeration that iterates over the given list. Of course, this resolves to nothing more than an invocation of the private constructor.

When testing, you will notice that it's cumbersome to have to repeatedly type `EnumerateItem`, particularly in the loop variable declaration.

```
List<String> ls = List.of("Nebraska", "Butterscotch", "Blackie", "Bella");
for (EnumerateItem item : EnumerateLazyList.enumerate(ls)) {
  System.out.println(item);
}
```

Java provides the `var` keyword to automatically infer the type of a variable, which makes the code significantly more concise.

```
var ls = List.of("Nebraska", "Butterscotch", "Blackie", "Bella");
for (var item : EnumerateLazyList.enumerate(ls)) {
  System.out.println(item);
}
```