Indiana University
CSCI-C212: Introduction to Software Systems (S25)
Joshua Crotts, Muazzam Siddiqui

Problem Set 5
April 2, 2025

Interfaces, Inheritance, Exceptions, File I/O

Important Dates:

- Assigned: April 2, 2025
- Deadline: April 23, 2025 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!**

What You Cannot Use:

There's not really much that you *can't* use for this problem set, since it involves you putting together (almost) everything that you have learned. Write clean, concise code that solves the problems.

Problem 1:

An *infinite lazy list* is one that, in theory, produces infinite results! We have illustrated this with Java's `Stream` API, but now we're going to design our own. Consider the `ILazyList` interface below:

```
interface ILazyList<T> {
  T next();
}
```

When calling `next` on a lazy list, we update the contents of the lazy list and return the next result. We mark this as a generic interface to allow for any desired return type. For instance, below is a lazy list that produces factorial values:[1]

```
class FactorialLazyList implements ILazyList<Integer> {

  private int n;
  private int fact;

  FactorialLazyList() {
    this.n = 1;
    this.fact = 1;
  }

  @Override
  public Integer next() {
    this.fact *= this.n;
    this.n++;
    return this.fact;
  }
}
```

Testing it with ten calls to `next` yields predictable results.

```
class FactorialLazyListTester {

  @Test
  void testFactorialLazyList() {
    ILazyList<Integer> FS = new FactorialLazyList();
    assertAll(
      () -> assertEquals(1, FS.next()),
      () -> assertEquals(2, FS.next()),
      () -> assertEquals(6, FS.next()),
      () -> assertEquals(24, FS.next()),
      () -> assertEquals(120, FS.next()));
  }
}
```

[1] We will ignore the intricacies that come with Java's implementation of the `int` datatype. To make this truly infinite (up to the system's memory limit), we could use `BigInteger`.

Design the `FibonacciLazyList` class, which implements `ILazyList<Integer>` and correctly overrides `next` to produce Fibonacci sequence values. You code should *not* use any loops or recursion. Recall that the Fibonacci sequence is defined as $f(n) = f(n-1) + f(n-2)$ for all $n \geq 2$. The base cases are $f(0) = 0$ and $f(1) = 1$.

```
class FibonacciLazyListTester {

  @Test
  void testFibonacciLazyList() {
    ILazyList<Integer> FS = new FibonacciLazyList();
    assertAll(
      () -> assertEquals(0, FS.next()),
      () -> assertEquals(1, FS.next()),
      () -> assertEquals(1, FS.next()),
      () -> assertEquals(2, FS.next()),
      () -> assertEquals(3, FS.next()),
      () -> assertEquals(5, FS.next()));
  }
}
```

Problem 2:

Design the `LazyListTake` class. Its constructor should receive an `ILazyList` and an integer *n* denoting how many elements to take, as parameters. Then, write a `List<T> getList()` method, which returns a `List<T>` of *n* elements from the given lazy list.

```
class LazyListTakeTester {

 @Test
 void testLazyListTake() {
  LazyListTake llt1 = new LazyListTake(new FactorialLazyList(), 8);
  LazyListTake llt2 = new LazyListTake(new FibonacciLazyList(), 10);

  assertAll(
    () -> assertEquals("[1, 2, 6, 24, 120, 720, 5040, 40320]",
                    llt1.getList().toString()),
    () -> assertEquals("[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]",
                    llt2.getList().toString()));
 }
}
```

Problem 3:

Java's functional API allows us to pass lambda expressions as arguments to other methods, as well as method references. Design the generic `FunctionalLazyList` class to implement `ILazyList`, whose constructor receives a unary function `Function<T, T> f` and an initial value `T t`. Then, override the `next` method to invoke f on the current element of the lazy list and return the previous. For example, the following test case shows the expected results when creating a lazy list of infinite positive multiples of three.

```
class FunctionalLazyListTester {

  @Test
  void testMultiplesOfThreeLazyList() {
    ILazyList<Integer> mtll = new FunctionalLazyList<>(x -> x + 3, 0);
    assertAll(
      () -> assertEquals(0, mtll.next()),
      () -> assertEquals(3, mtll.next()),
      () -> assertEquals(6, mtll.next()),
      () -> assertEquals(9, mtll.next()),
      () -> assertEquals(12, mtll.next()));
  }
}
```

What's awesome about this exercise is that it allows us to define the elements of the lazy list as any arbitrary lambda expression, meaning that we could redefine `FactorialLazyList` and `FibonacciLazyList` in terms of `FunctionalLazyList`. We can generate infinitely many ones, squares, triples, or whatever else we desire.

Problem 4:

Design the generic `CyclicLazyList` class, which implements `ILazyList`, whose constructor is variadic and receives any number of values. Upon calling `next`, the cyclic lazy list should return the first item received from the constructor, then the second, and so forth until reaching the end. After returning all the values, cycle back to the front and continue. For instance, if we invoke `new CyclicLazyList<Integer>(1, 2, 3)`, invoking `next` five times will produce `1`, `2`, `3`, `1`, `2`.

Problem 5:

(Note about this exercise: it is long, involved, and accounts for 25% of your PSet5 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!)

In this series of problems, you will design several methods that act on very large/small *integers* resembling the `BigInteger` class. You ***cannot*** use any methods from `BigInteger`, or the `BigInteger` class itself.

(a) Design the `BigInt` class, which has a constructor that receives a string. The `BigInt` class stores a `List<Integer>` as an instance variable, as well as a boolean for keeping track of whether it is negative. You will need to convert the given string into said list of digits. Store the digits in reverse order, i.e., the least-significant digit (the ones digit) of the number is the first element of the list. Leading zeroes should be omitted from the representation.

The possible values for the parameter are an optional sign (either positive **or** negative), followed by one or more digits.

Below are some example inputs.

```
new BigInt("42")         => [2, 4], isNegative = false
new BigInt("0420")       => [0, 2, 4], isNegative = false
new BigInt("-42")        => [2, 4], isNegative = true
new BigInt("0000420000") => [0, 0, 0, 0, 2, 4], isNegative = false
new BigInt("+42")        => [2, 4], isNegative = false;
```

(b) Override the `public boolean equals(Object o)` method to return whether `this` instance represents the same integer as the parameter. If `o` is not an instance of `BigInt`, return `false`.

```
new BigInt("42").equals(new BigInt("42"))      => true
new BigInt("00042").equals(new BigInt("0042")) => true
new BigInt("+42").equals(new BigInt("42"))     => true
new BigInt("42").equals(new BigInt("-42"))     => false
new BigInt("-42").equals(new BigInt("42"))     => false
new BigInt("422").equals(new BigInt("420"))    => false
```

(c) Override the `public String toString()` method to return a stringified version of the number. Remember to include the negative sign where appropriate. If the number is positive, you do not need to include it.

(d) Implement the `Comparable<BigInt>` interface and override the method it provides, namely `public int compareTo(BigInt b2)`. Return the result of comparing `this` instance with the parameter. That is, if $a < b$, return -1, if $a > b$, return 1, and otherwise return 0, where a is `this` and b is `b2`.

(e) Design the `BigInt copy()` method, which returns a (deep)-copy of instance representing the same integer as `this` instance of `BigInt`. Do *not* simply copy the reference to the list of digits over to the new `BigInt` (this violates the aliasing principle that we have repeatedly discussed and can be the cause of relentless debugging).

(f) Design the `BigInt negate()` method, which returns a copy of `this` instance of `BigInt`, but negated. Do *not* modify `this` instance.

(g) Design the `private boolean areDifferentSigns(BigInt b)` method, which returns whether `this` instance and `b` have different signs. That is, if one is positive and one is negative, `areDifferentSigns` returns `true`, and false otherwise.

(h) Design the `private BigInt addPositive(BigInt b)` method, which returns a `BigInt` instance that is the sum of `this` and `b` under the assumption that `this` and `b` are non-negative. *Be sure you thoroughly test this method!*

(i) Design the `private BigInt subPositive(BigInt b)` method, which returns a `BigInt` instance that is the difference of `this` and `b` under the assumption that `this` and `b` are non-negative, and the minuend (the left-hand operand) is greater than or equal to the subtrahend (the right-hand operand). *Be sure you thoroughly test this method!*

(j) Design the `private BigInt mulPositive(BigInt b)` method, which returns a `BigInt` instance that is the product of `this` and `b` under the assumption that `this` and `b` are non-negative. *Be sure you thoroughly test this method!*

(k) Design the `BigInt add(BigInt b)` and `BigInt sub(BigInt b)` method that returns the sum/difference of `this` and `b` respectively. Note that these methods should be a case analysis of the signs of the operands. Use the following equivalences to guide your design. Do *not* over-complicate these methods.

$$
\begin{aligned}
A + (-B) &= A - B \text{ if } A \geq B. \text{ Otherwise, } -(B - A).\\
(-A) + (-B) &= -(A + B).\\
A - (-B) &= A + B.\\
(-A) - B &= -(A + B).\\
(-A) - (-B) &= (-A + B) \text{ if } A \geq B. \text{ Otherwise, } (B - A).
\end{aligned}
$$

(l) Design the `BigInt mul(BigInt b)` method that returns the product of `this` and `b`. The product of two negative integers is a positive integer, and the product of exactly one positive and exactly one negative is a negative integer.

(m) **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.

Problem 6:

Design the `Capitalize` class, which contains one `static` method: `void capitalize(String in)`. The `capitalize` method reads a file of sentences (that are not necessarily line-separated), and outputs the capitalized versions of the sentences to a file of the same name, just with the `.out` extension (you must remove whatever extension existed previously).

You may assume that a sentence is a string that is terminated by a period and only a period, which is followed by a single space. If you use a splitting method, e.g., `.split`, you must remember to reinsert the period in the resulting string. There are many ways to solve this problem!

Example Run. If we invoke `capitalize("file2a.in")` into the running program, and `file2a.in` contains the following ***(note that if you copy and paste this input data, you will need to remove the newline before the `"hopefully"` token):***

```
hi, it's a wonderful day. i am doing great, how are you doing. it's
hopefully fairly obvious as to what you need to do to solve this problem.
this is a sentence on another line.
this sentence should also be capitalized.
```

then `file2a.out` is generated containing the following (***again, remember to remove the newline before `"hopefully"`.***):

```
Hi, it's a wonderful day. I am doing great, how are you doing. It's
hopefully fairly obvious as to what you need to do to solve this problem.
This is a sentence on another line.
This sentence should also be capitalized.
```

Problem 7:

Design the `SpellChecker` class, containing the `static void spellCheck(String dict, String in)`. The `spellCheck` method reads two files: a "dictionary" and a content file. The content file contains a single sentence that may or may not have misspelled words. Your job is to check each word in the file and determine whether or not they are spelled correctly, according to the dictionary of (line-separated) words. If a word is not spelled correctly, wrap it inside brackets `[]`.

Output the modified sentences to a file of the same name, just with the `.out` extension (you must remove whatever extension existed previously). You may assume that words are space-separated and that no punctuation exists. Hint: use a `Set`! Another hint: words that are different cases are not misspelled; e.g., `"Hello"` is spelled the same as `"hello"`; how can your program check this?

Example Run. Assuming `dictionary.txt` contains a list of words, if we invoke the method with `spellChecker("dictionary.txt", "file3a.in")`, and `file3a.in` contains the following:

```
Hi hwo are you donig I am dioing jsut fine if I say so mysefl but I
will aslo sya that I am throughlyy misssing puncutiation
```

then `file3a.out` is generated containing the following:

```
Hi [hwo] are you [donig] I am [dioing] [jsut] fine if I say so
[mysefl] but I will [aslo] [sya] that I am [throughlyy] [misssing]
[puncutiation]
```

Problem 8:

A maze is a grid of cells, each of which is either open or blocked. We can move from one free cell to another if they are adjacent. Design the `MazeSolver` class, which has the following methods:

(a) `MazeSolver(String fileName)` is the constructor, which reads a description of a maze from a file. The file contains a grid of characters, where `'.'` represents an open cell and `'#'` represents a blocked cell. The file is formatted such that each line is the same length. Read the data into a `char[][]` instance variable. You may assume that the maze dimensions are on the first line of the file, separated by a space.

(b) `char[][] solve()` returns a `char[][]` that represents the solution to the maze. The solution should be the same as the input maze, but with the path from the start to the end marked with `'*'` characters. The start is the top-left cell, and the end is the bottom-right cell. If there is no solution, return `null`.

We can use a backtracking algorithm to solve this problem: start at a cell and mark it as visited. Then, recursively try to move to each of its neighbors, marking the path with a `'*'` character. If you reach the maze exit, then return `true`. Otherwise, backtrack and try another path. By "backtrack," we mean that you should remove the `'*'` character from the path. If you have tried all possible paths from a cell and none of them lead to the exit, then return `false`. We provide a skeleton of the class below.

(c) `void output(String fileName, char[][] soln)` outputs the given solution to the maze to a file specified by the parameter. Refer to the above description for the format of the output file and the input `char[][]` solution.

Extra Credit (10 points):

(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.

Extra Credit (10 points):

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);
}
```