Indiana University
CSCI-C212: Introduction to Software Systems (SP24) Assignment 5
Joshua Crotts & Muazzam Siddiqui March 27, 2024

Interfaces, Inheritance, Exceptions, File I/O

Important Dates:

- Assigned: April 3, 2024
- Deadline: April 17, 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.

Problem 1

A *lazy list* is one that, in theory, produces infinite results! 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 int next() {
    this.fact *= this.n;
    this.n++;
    return this.fact;
  }
}
```

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

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

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, we could use `BigInteger`.

```
      () -> assertEquals(720, FS.next()),
      () -> assertEquals(5040, FS.next()),
      () -> assertEquals(40320, FS.next()),
      () -> assertEquals(362880, FS.next()),
      () -> assertEquals(3628800, FS.next()));
  }
}
```

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 $f(1) = 1$.

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

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()),
      () -> assertEquals(8, FS.next()),
      () -> assertEquals(13, FS.next()),
      () -> assertEquals(21, FS.next()),
      () -> assertEquals(34, FS.next()));
  }
}
```

Problem 2

Design the `LazyListTake` class. It 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.

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

class LazyListTakeTester {

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

  assertAll(
    () -> assertEquals("[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]",
                       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 (as we saw in Chapter 5). Design the generic `FunctionalLazyList` class to implement `ILazyList`, whose constructor receives a unary function `Function<I, R> 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.

```
import static Assertions.assertEquals;
import static Assertions.assertAll;

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 `FunctionLazyList`. 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

This exercise is multi-part and involves the interpreter we wrote in the chapter. Please reference the starter code posted in Canvas.

(a) First, design the `ProgramNode` class, which allows the user to define a program as a sequence of statements rather than a single expression.

(b) Design the `DefNode` class, which allows the user to create a global definition. Because we're now working with definitions that do not extend the environment, we should use the set method in environment. When creating a global definition via `DefNode`, we're expressing the idea that, from that point forward, the (root) environment should contain a binding from the identifier to whatever value it binds.

(c) Design the `FuncNode` node. We will consider a function definition as an abstract tree node that begins with `FuncNode`. This node has two parameters to its constructor: a list of parameter (string) identifiers, and a single abstract syntax tree node representing the body of the function. We will only consider functions that return values; void functions do not exist in this language.

(d) Design the `ApplyNode` class, which applies a function to its arguments. You do not need to consider applications in which the first argument is a non-function.

Calling/Invoking a function is perhaps the hardest part of this exercise. Here's the idea, which is synonymous and shared with almost all programming languages:

(i) First, evaluate each argument of the function call. This will result in several l-values, which should be stored in a list.

(ii) Convert these to their AST counterparts.

(iii) We then want to create an environment in which the formal parameters are bound to their arguments. Overload the extend method in `Environment` to now receive a list of string identifiers and a list of (evaluated) AST arguments. Bind each formal to its corresponding AST, and return the extended environment.

(iv) Evaluate the function identifier to get its function definition and convert it to an AST.

(v) Call `eval` on the function body and pass the new (extended) environment.

This seems like a lot of work (because it is), but it means you can write really cool programs, including those that use recursion!

```
new ProgramNode(
  new DefNode("!",
    new FuncNode(
      List.of("n"),
      new IfNode(
        new PrimNode("eq?",
          new VarNode("n"),
          new NumberNode(0)),
        new NumberNode(1),
        new PrimNode("*",
          new VarNode("n"),
          new ApplyNode("!",
            new PrimNode("-",
            new VarNode("n"),
            new NumberNode(1)))))))),
  new ApplyNode("!", new NumberNode(5)))
```

Problem 6

In this series of exercises, you will design several methods that act on very large natural numbers resembling the `BigInteger` class. You cannot use any methods from the class, or the class itself. In this problem you will design several methods that act on very large *natural numbers* resembling the `BigInteger` class. You ***cannot*** use any methods from this class, or the class itself.

(a) Design the `BigNat` class, which has a constructor that receives a string. The `BigNat` class stores a `List<Integer>` as an instance variable. You will need to convert the given string into said list. 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.

(b) Override the `public String toString()` method to return a string representation of the `BigNat` object.

(c) Override the `BigNat clone()` method that returns a new `BigNat` instance that contains the same number.

(d) Override the `public boolean equals(Object obj)` method to compare two `BigNat` values for equality. Remember that you have to cast the given parameter to an instance of the `BigNat` class.

(e) Implement the `Comparable<BigNat>` interface, and override the `public int compareTo(BigNat b)` method to return the sign of the result of comparing the given `BigNat` (which we will call b) to `this BigNat` (which we will call a). Namely, if $a < b$, return -1, if $a > b$, return 1, otherwise return 0.

(f) Design the `void add(BigNat bn)` method, which adds a `BigNat` to `this BigNat`. The method should not return anything. Note: this problem is harder than it may look at first glance!

(g) Design the `void sub(BigNat bn)` method, which subtracts a `BigNat` from `this BigNat`. If the subtrahend (the right-hand side of the subtraction) is greater than the minuend, the result is zero. Over natural numbers, this is called the *monus* operator.

(h) Design the `void mul(BigNat bn)` method, which multiplies a `BigNat` with `this BigNat`. Note: remember how we implement multiplication recursively? You shouldn't use recursion for this problem, but what *is* multiplication? Think about the performance implications of this approach.

(i) Design the `void div(BigNat bn)` method, which divides a `BigNat` with `this` `BigNat`. If the divisor is greater than the dividend, assign the dividend to be zero. If the divisor is zero, do nothing at all. Otherwise, perform integer division. Note: we can implement division recursively. You shouldn't use recursion for this problem, but what *is* division? Think about the performance implications of this approach.

Problem 7

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 8

Design the `SpellChecker` class, which contains one `static` method: `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 exist. 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]
```