Programming Project 4

Due Sunday, December 5 at 11:59pm

As Monday, December 6 is a Reading Day and that day should to be spent preparing for final exams, *no late assignments will be accepted*.

IMPORTANT: *Read the* ***Do's and Dont's*** *in the* ***Course Honor Policy*** *found on blackboard.*

I. Overview

DNA

DNA is a molecule made up of a sequence of nucleobases: adenine (A), cytosine (C), guanine (G), and thymine(T). The sequence of bases in the DNA form the code of our genes. You may have heard that scientists are sequencing (determining the order of the bases) the DNA from a large number of species, including humans. To sequence DNA from a sample, the sample needs to contain many copies of the DNA. For example, the sample could contain a large number of human cells from the same person. Then we do the following:

1. the multiple copies of the DNA are chopped up into small pieces
2. the pieces are put into a solution with appropriate molecules so that the DNA pieces are cloned into many, many copies
3. sequencer machines can identify the DNA sequence of the small pieces
4. then it becomes a giant puzzle problem as programs are used to figure out how to reassemble the little pieces into the original long DNA strands.

If we have enough copies of the original DNA, then when we break them each up into pieces, we will not break the different copies at the same spots. The sequencing programs use this overlap between pieces as the clue for determining which pieces connect to which pieces.

A Short Explanation of Enum

You will use the `enum` type in this project.

An *enum* is a shortcut for a class with a private constructor. The code

```
enum WeekDay {
  Monday, Tuesday, Wednesday, Thursday, Friday;
  you may add additional methods here
}
```

is identical to

```
public static class WeekDay {
  public static final WeekDay Monday = new WeekDay();
  public static final WeekDay Tuesday = new WeekDay();
  public static final WeekDay Wednesday = new WeekDay();
  public static final WeekDay Thursday = new WeekDay();
  public static final WeekDay Friday = new WeekDay();

  private WeekDay() {
  }

  some special helper methods provided for enums (see your text)
```

```
  you may add additional methods here
}
```

So, `WeekDay` will be static inner classes of whatever class you place the enum inside. The `Monday`, `Tuesday`, etc. are fields set to instances of the `WeekDay` class. Because the constructor is private, no other instances can be created than one instance for each of the listed fields. (Note that we do not need to override the `equals` method in an `enum`. Since no other instances can be created than those stored in the fields, you can use `==` to compare enum values.) As entered, you have a default private constructor for the enum, but you can create your own constructor if you wish. For example, we could create a constructor that takes a `String` that is the name of the day. If we do that, we would need to do the following:

```
enum WeekDay {
  Monday("Monday"), Tuesday("Tuesday"), Wednesday("Wednesday"), Thursday("Thursday"), Friday("Friday");

  private String name;

  private WeekDay(String name) {
    this.name = name;
  }

  you may add additional methods here
}
```

II. Code Readability (20% of your project grade)

To receive the full readability marks, your code must follow the following guideline:

- All variables (fields, parameters, local variables) must be given appropriate and descriptive names.
- All variable and method names must start with a lowercase letter. All class names must start with an uppercase letter.
- The class body should be organized so that all the fields are at the top of the file, the constructors are next, the non-static methods next, and the static methods at the bottom with the main method last.
- There should not be two statements on the same line.
- All code must be properly indented (see page 689 of the Lewis book for an example of good style). The amount of indentation is up to you, but it should be at least 2 spaces, and it must be used consistently throughout the code.
- You must be consistent in your use of {, }. The closing } must be on its own line and indented the same amount as the line containing the opening {.
- There must be an empty line between each method.
- There must be a space separating each operator from its operands as well as a space after each comma.
- There must be a comment at the top of the file that **is in proper JavaDoc format** and includes both your name and a description of what the class represents. The comment should include tags for the author.
- There must be a comment directly above each method (including constructors) that **is in proper JavaDoc format** and states *what* task the method is doing, not how it is doing it. The comment should include tags for any parameters, return values and exceptions, and the tags should include appropriate comments that indicate the purpose of the inputs, the value returned, and the meaning of the exceptions.
- There must be a comment directly above each field that, in one line, states what the field is storing.
- There must be a comment either above or to the right of each non-field variable indicating what the variable is storing. Any comments placed to the right should be aligned so they start on the same column.
- There must be a comment above each loop that indicates the purpose of the loop. Ideally, the comment would consist of any preconditions (if they exist) and the subgoal for the loop iteration.
- Any code that is complicated should have a short comment either above it or aligned to the right that explains the logic of the code.

III. Program Testing (20% of your project grade)

You are to create two JUnit test classes: `DoubleLinkedListTester` and `DNATester`. All your tests should be coded as unit tests in these classes. Your report should be a short document that explains why the JUnit tests you chose to do thoroughly test each method.

IV. Java Programming (60% of your grade)

You will write a DNA sequencing program.

General Guidelines

VERY IMPORTANT: The purpose of this homework is to give you practice using lists, generic types, and iterators. You must use these types correctly.

- You must *NOT* have any warning messages when compiling your code due to improper use of generic types.
- In many places, you are given the choice of using the linked list from class or the one from the Java API. However, you *MUST* use the types correctly. Just because a method exists does not mean you can use it. In particular, you should not access elements of the linked list by index nor do other operations that require extra traversals of the linked list.
- You need to avoid unnecessary fields in your class.
- Your loops should be simple. Avoid nested loops unless they are required. Use good loop conditions so that you do not need `break` or `continue` or break-like code in the body of your loops.

Step 1: Add methods to `DoubleLinkedList`.

Add the following methods to the `DoubleLinkedList` class you started in lab. (At the minimum, you should have completed the `addToFront` and `addToBack` methods.)

1. `equals`: This method should override the `equals` method of `Object`. Two double linked lists are equal if they contain the same elements in the same order. **Your method should not have any warning messages when compiled.** *Hint: Use the @Override annotation so you do not accidentally make a simple mistake.*
2. `append`: a `void` method that takes a `DoubleLinkedList` and appends the nodes of the linked list to the end of the nodes of this list. The double linked list parameter may be destroyed by this method.

Step 2: Make `DoubleLinkedList` implement the `Iterable` interaface.

You are to expand the `DoubleLinkedList` class you worked on in lab. You are to make the `DoubleLinkedList` implement `Iterable` and have the `iterator` method return a `ListIterator` from the API. `ListIterator` is an extension of `Iterator` and has additional method stubs that must be overridden. You are required to implement the methods

1. `add`
2. `hasNext`
3. `hasPrevious`
4. `next`
5. `previous`
6. `set`

Your implementations *must* match the descriptions given in the Java API. You *may* implement the other methods of the interface, but that is not required. For any method you choose to not implement, have the method throw a

`UnsupportedOperationException`.

Step 3: Create a `DNA` class

Create a class `DNA` that contains a public *enum* type called `Base`. The `Base` type should have four possible values: `A`, `C`, `G`, and `T`. The four letters represent the four different amino acids that make up a DNA sequence.

Organize your class so `DNA` is a `DoubleLinkedList` that can only store elements of `Base`. You may use the `LinkedList` class from the API instead of your `DoubleLinkedList` class. **However**, you must still use the linked list in an efficient and appropriate manner. Just because a method exists in the API does not mean you should use it.

Add the following methods to the `DNA` class:

1. `String toString()`: returns a `String` representation of the DNA. The string should be the letters (without spaces) representing the amino acids. For example, `"ACGGCGT"` represents a DNA that contains 7 bases. The base at the *head* of the DNA is A and the base at the *tail* is T. *For this method, you should use a for-each loop to run through the DNA bases.*
2. `static DNA string2DNA(String s)` given a String, returns a the DNA sequence represented by the String. For example, given the string `"GCGTTATA"`, the method should return a DNA with 8 bases, the head base is G and the tail base is A. If the `String s` is empty or contains a character that is not G, T, C, or A, throw an `IllegalArgumentException`.
3. `void splice(DNA dna, int numbases)`: remove `numbases` from the start of `dna` and then append the remaining bases of `dna` to the end of this `DNA`. If `dna` has fewer than `numbases` bases, then nothing is appended. The parameter `dna` may be destroyed by this method.
4. `static boolean overlaps(DNA dna1, DNA dna2, int n)`: returns `true` if the last `n` bases of `dna1` exactly match the first `n` bases of `dna2`. Returns `false` if there is no such match.
5. `main`: Takes two strings that represent DNA sequences, determines the greater overlap, the end of the first to the start of the second or the end of the second to the start of the first. The method then performs the appropriate splicing to create the minimum DNA sequence that splices the two strings and prints the result. The `main` method should not throw any exceptions. Instead, if bad data is entered, the method should print an appropriate message.

   For example, if we run the program with `java DNA CGCTCACCTAT ATAATCGCTC`, then the largest overlap of the first onto the second is 2 bases (AT), but the largest overlap of the second onto the first is 5 bases (CGCTC). We would then splice the second onto the first producing ATAATCGCTCACCTAT.

Make the DNA `Comparable`. A shorter DNA strand should be ordered before a longer DNA strand. Two strands with equal lengths should be ordered alphabetically. You should be able to implement this using a *single* traversal of the data.