intro2cs/introcsp/introcst

Ex8: Interfaces

In this exercise, you will practice implementing interfaces and using object composition.
You will create a program that can parse arbitrary strings that represent sequences of natural numbers, and display them in a window.

Part 1: Basic Sequence Classes

Have a look at the Sequence interface, which we provide you with. This interface represents a sequence of real numbers, whose elements are returned one-by-one, using an iterator-style API. A sequence may be finite or infinite. Your job in the first part of the exercise is to write several simple classes that implement the Sequence interface, which are described below. When implementing these classes, pay attention to special notes in the class APIs regarding the size of each sequence and its behavior if division by zero is required, etc.

Please note: throughout the exercise, whenever we mention "the *n*'th element" of a sequence, we are referring to the n'th element in the sequence when counting elements in order starting with the 0'th element.

Primitive Sequences

The most basic type of a sequence is the so-called identity sequence (IdentitySequence), in which the *n*'th element is simply the natural number *n* (for *n*=0,1,2,...). Note that conceptually, this is an infinite sequence. Another type of sequence is the finite sequence (FiniteSequence), which consists of finitely many numbers (not necessarily following any pattern), explicitly provided to the constructor. Other basic sequences include the infinite sequence whose elements are all equal to the same constant number (ConstantSequence), and the Fibonacci sequence (FibonacciSequence).

Compound Sequences

We move on to several "compound sequences" - sequences that are themselves defined using one or more objects that implement the Sequence interface:

Let *T=(T0,T1,T2,...)* and *Q=(Q0,Q1,Q2,...)* be some sequences (i.e. some instances of classes that implement the Sequence interface).

Define the ratio sequence (ConsecutiveRatioSequence) of *T* to be the sequence *R* such that the *n*'th element in *R* is *(T(n+1) / T(n))*. To avoid division errors, if T(n) == 0, we define the n'th element of R to be 0, as if the result of a division by zero were zero.
Define the sum sequence (CumulativeSumSequence) of *T* to be the sequence *S* such that the *n*'th element in *S* is the sum of *(T0 + T1 + … + Tn)*.
Define the addition sequence (AdditionSequence) of *T* and *Q* to be the sequence *A* such that the *n*'th element in *A* is *(Tn + Qn)*.
Define the division sequence (DivisionSequence) of *T* and *Q* to be the sequence *D* such that the *n*'th element in *D* is *(Tn / Qn)*. Once again, if Qn is zero, then the n'th element of D is defined to be zero, as if the result of a division by zero were zero.
Define the multiplication sequence (MultiplicationSequence) of *T* and *Q* to be the sequence *M* such that the *n*'th element in *M* is *(Tn * Qn)*.
Define the power sequence (PowerSequence) of *T* and *Q* to be the sequence *P* such that the *n*'th element in *P* is *(Tn ** Qn)*, where ** is the power operator.

Part 2: The Sequence Factory

In this part of the exercise you will implement a SequenceFactory class that contains one static function that creates sequences: **`static Sequence sequenceFromString(String)`**
This function is given a string, for example "`(n^2)/2`", and returns a Sequence whose elements are obtained by substituting all the natural numbers (*n=0*, *1, 2, …*) in place of *n* in the given expression.
Note that for different values of the given string, the actual type (class) of the return value of sequenceFromString will be different, but it will always implement the Sequence interface. This is the reason we implement this functionality as a method, and not as some constructor (as each constructor only knows how to construct a single class).
For the example above, the sequence elements will be (0, 0.5, 2, …). Note that the sequence conceptually never ends, as it is defined for all natural numbers.

As is defined in the SequenceFactory javadoc, the input to the factory method is assumed to contain a legal sequence string, possibly padded with whitespace characters.

Expression Strings

To define legal input for our factory function, we formally define what we consider to be a legal expression string. An expression string is defined recursively:

- A double literal is an expression.
- The variable name "`n`" is an expression.
- The Fibonacci sequence name "`fib()`" is an expression.
- If "`E`" is an expression, then "`(E)`" is an expression.
- If "`E`" is an expression, then "`-E`" is an expression.
- If "`E`" is an expression, then "`ratio(E)`" is an expression.
- If "`E`" is an expression, then "`sum(E)`" is an expression.

- If “E1” and “E2” are both expressions, then each of the following is also an expression:
    - “E1+E2”
    - “E1-E2”
    - “E1*E2”
    - “E1/E2”
    - “E1^E2”

Note that the empty string “” is **not** considered a legal expression.

Parsing Expressions Into Sequences

Although we now have an exact syntax of what an expression is, we also must define rules on how an expression is parsed. This is necessary as many times an expression’s meaning may be ambiguous: without any operator precedence rules, we could parse the string “1+2/4” in two ways: (1+2)/4=0.75, or 1+(2/4)=1.5. Clearly, as is done in algebra, in our program too the rules of precedence must be formally defined when parsing a sequence string.

The logic that we will use to parse expressions into sequences is as follows, given some input string that represents an expression (note that the parsing is done recursively, following the recursive definition given earlier):

Priority is given using the following rules (**in this order**), recursively:

1. Any operators inside parenthesized parts are ignored until later recursive calls.
    - For example, in an input "3*(2+4)", the '+' operator is irrelevant and ignored until a recursive call is made on the substring “2+4”. The only relevant operator in this example is '*' since it is not inside parentheses.
2. If the input is "expr1+expr2" or "expr1-expr2", recursively return the addition sequence "expr1"+"expr2" or "expr1"+"-expr2", respectively.
    - Note: any '-' signs that appear directly to the right of another operator, or are at the left-most index of the text, are ignored when applying this rule. In the following examples the '-' sign is ignored until later recursive calls: "-5*n", "1/-n".
    - If more than one relevant '+' or '-' symbols appear, choose the last (right-most) symbol among them. See example in the notes section.
3. If the input is "expr1*expr2" or "expr1/expr2", recursively return the multiplication sequence "expr1"*"expr2" or the division sequence "expr1"/"expr2", respectively.
    - If more than one relevant '*' or '/' symbols appear, choose the last (right-most) symbol among them. See example in the notes section.
4. If the input is "-expr", recursively return the multiplication sequence of "expr" by the constant sequence -1.
5. If the input is "expr1^expr2", recursively return the power sequence "expr1"^"expr2".
    - If more than one relevant '^' symbols appear, choose the last (right-most) symbol among them. See example in the notes section.

In case none of the above applies, do the following:

1. If the input is "`(expr)`", recursively return the sequence corresponding to "`expr`".
2. If the input is "`n`", return an identity sequence.
3. If the input is "`fib()`", return a Fibonacci sequence.
4. If the input is "`ratio(expr)`", recursively return the consecutive ratio sequence of "`expr`".
5. If the input is "`sum(expr)`", recursively return the commulative sum sequence of "`expr`".
6. Else, the input is a number; return a constant sequence accordingly. The function `Double.parseDouble()` may be used here.

Part 3: The Sequence Viewer

In this part, you will implement the SequenceViewer class, which contains a `main` method. This class reads an input text file that describes one or more sequences, and graphically plots them in a window. In addition, the input file's contents may dictate that the display window will emphasize one or more points (for aesthetic purposes only), which we shall name "limit points". Displaying the limit points and the sequences is a straight-forward process of passing them to the plotting window functions, as you shall see in the API of the plotting class that we provide you with.

When implementing this class, you should use the SequencePlotter class which is provided in the intro package. You should also use the IntroUtil.newScannerFromFile method (which you've already used in ex5) in order to read input text files.

The SequencePlotter class, which we supply you with, supports two main operations: plotting a Sequence object to the screen using a given color and style, and displaying a "limit point" on the far right of the screen. The limit point functionality is meant for aesthetic purposes only; it simply displays a small dot at a given Y value, on the right side of the window. Suppose, for example, that you plot the sequence "`(5*n^2 + 7*n)/n^2`" that is known (mathematically) to have a limit at y = 5, as *n* tends to infinity; in this case, you can add a limit point at 5.0 to the plotter, and this lets you see more easily that the sequence indeed converges to that number, as *n* attains larger and larger values. We shall state once again that this limit point functionality is for aesthetic purposes only - it allows you to emphasize a certain Y value on the 2D plane, which makes sense when dealing with series that have limits at infinity.

The program you will implement receives exactly one argument. In case the number of arguments is not 1, you should print the following string and terminate the program gracefully:

```
Usage: SequenceViewer filePath, where filePath is a path to an input file.
```

You may safely assume that when the input arguments contain one parameter, it is a legal path to a file that exists and is accessible in the filesystem.

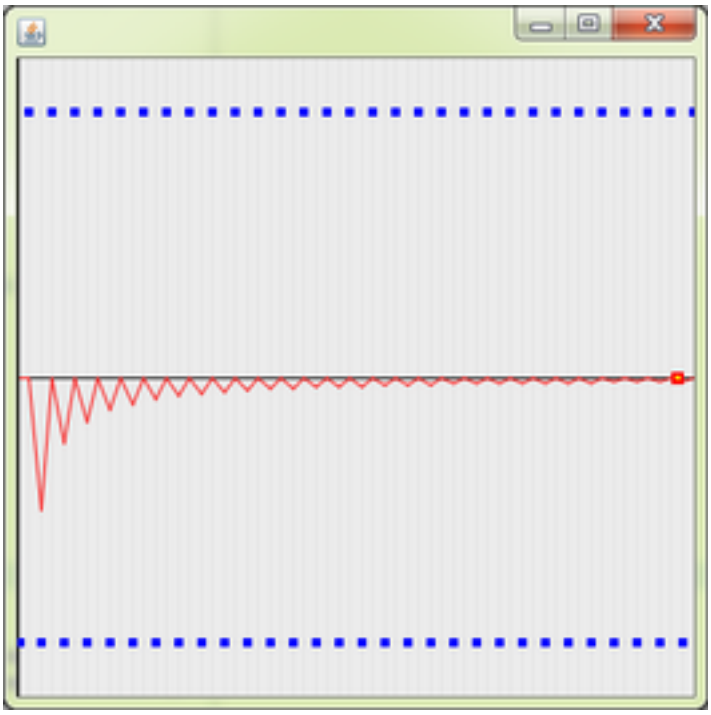
First, we note that any whitespace characters in the input file may be safely ignored. The input file may contain any number of lines, and each line may be of the following types:

- A “limit point” line: these lines are of the form “LIMIT_POINT, double”. Such a line indicates that the given double value should be used as a limit point in the plotter window.
    - For example: `LIMIT_POINT, 5`
- A sequence definition line: these lines are of the form “SEQUENCE, color, style, number of points to display, expression string”.
    - For example (in the file, this will be one line and not two): `SEQUENCE, green, CONTINUOUS_LINEAR_INTERPOLATION, 1000, (5*n^2 + 7*n)/n^2`
    - The color may be one of `black`, `red`, `green`, `blue`. Please use the constant colors defined in java.awt.Color.
    - The style may be one of `DOT_PLOT` or `CONTINUOUS_LINEAR_INTERPOLATION`. Please see the SequencePlotter.CURVE_DISPLAY_STYLE enumeration.
    - The number of points to display tells the plotter how many points should be used to generate the plot. The sequence values serve as the Y values of these points. The first point’s X value is 0, the second point’s X value is 1, and so on.
    - The last item on a sequence definition line is assumed to be a legal sequence expression string. It should be parsed into a Sequence object using your SequenceFactory class and passed to the SequencePlotter along with the other parameters on the line.
- Empty lines (or lines that contain only whitespace characters) may be ignored.

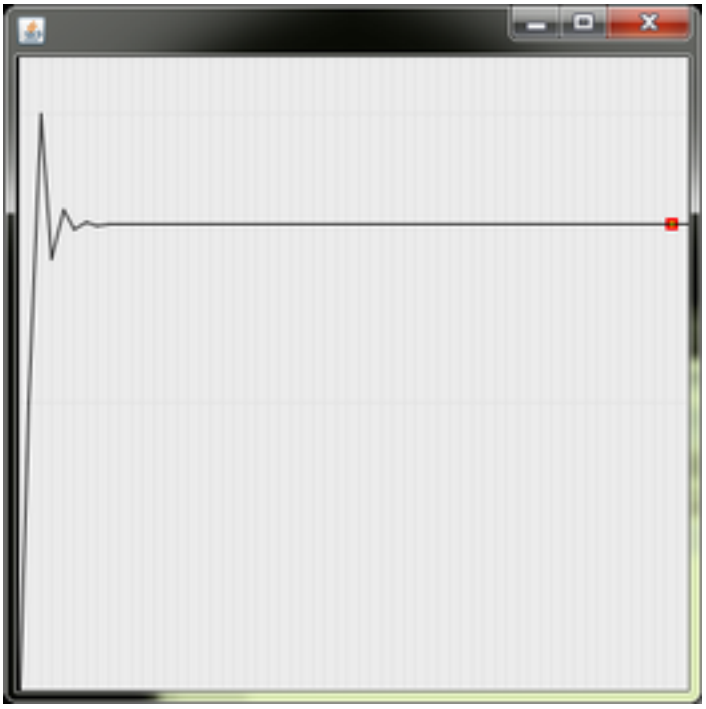
Sample Output

Try and think about a nice example of your own for displaying the properties of sequences and their limits. Explain what’s interesting in your example in the README file, and include in your submission a file called `myInput.txt` which presents it. Don’t worry If you can’t think of a particularly interesting example; we only require that you have at least one function and one limit point in the file that you submit.

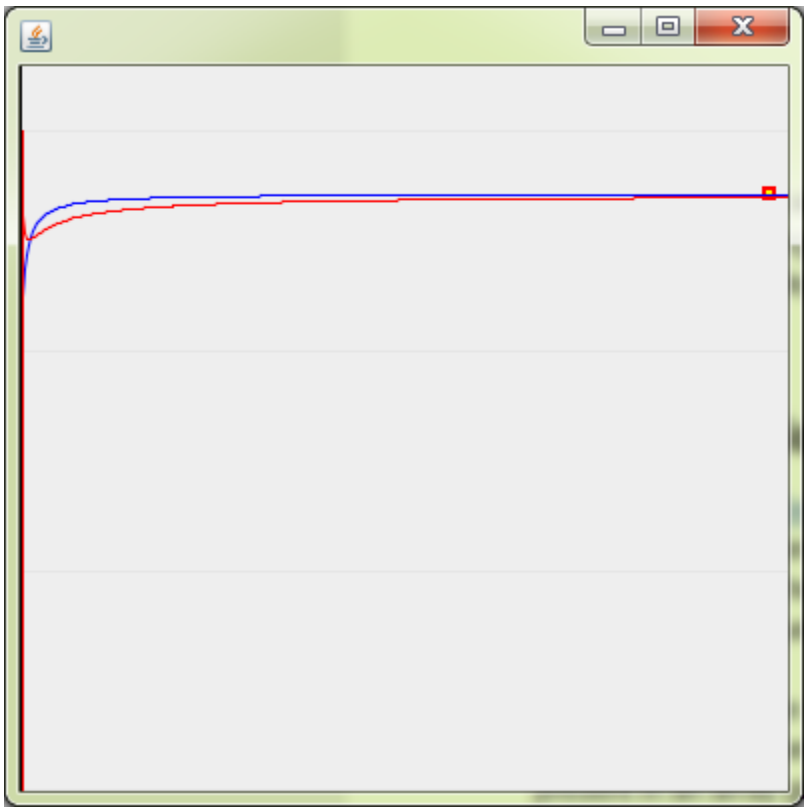
The following are sample outputs for the main method, and short explanations on why we consider them interesting. The input files for these examples may be downloaded here.



In dotted blue, we see the sequence whose elements are -1,1,-1,1,...
In red, we see the Cesaro mean sequence of the blue sequence.
We can see clearly that the mean sequence has a limit at Y=0.



In this plot, we see the sequence whose elements are the ratio of consecutive elements in the Fibonacci series. That is, the sequence plotted is `ratio(fib())`. It is widely known that the limit of this sequence is the so-called "golden ratio" number, which is approximately 1.618.



In blue, we see the sequence whose elements are `(1 + (1/n))^n`. The limit of this series is Euler's constant, e.
In red, we see the Cesaro mean sequence of the red sequence.
It is easy to prove that if a series has a limit at infinity, then the Cesaro mean sequence of that series also has a limit at infinity, and the limits are equal. As can be seen in this plot, the mean series indeed also converges to Euler's constant, as expected.

Notes

Each operator is defined to be in a precedence group: the precedence groups are {`+`, `-`}, {`*`, `/`}, {`^`}. As can be inferred from the rules above, whenever there is more than one relevant symbol from the same precedence group (from the precedence group with highest priority among all those present), we always choose to use the last (right-most) symbol in the input. For example, the input "`2*10/2`" is parsed as (2*10)/(2) because '`*`' is to the left of '`/`'. Similarly, "`2/10*2`" is parsed as (2/10)*(2).

You may safely assume that during testing, we will never call the `next()` function at a time when `hasNext()` would return a value of `false`.

When parsing the input file's text, you may find String's split() method to be very useful: given a single line from the input file, splitting it according to ',' characters will let you access the various parts of the line string easily.

In order to remove all whitespace from a given string, you may find the String method replaceAll() useful. By calling `str.replaceAll("\\s", "")` on some string `str`, we tell the string to return a new instance in which all substrings matching the *Java regular expression* `\\s` are replaced with an empty string (i.e. removed). *Regular expressions* in Java (not to be confused with the "expressions" that we defined in this exercise!) are a very strong tool that can be helpful in many tasks, which we haven't learned in this course. For now it suffices to say that `\\s` represents "any whitespace character", i.e. spaces, tabs, carriage returns, etc.

We will not test your code in cases where NaN or infinity values occur. Behavior of your classes for these values is undefined.

The toString() method of various sequence classes is defined explicitly. Make sure you implement this method according to its definition for each class. A helpful tip: you can use this method to help debug your code.

Documentation

In this exercise, as usual, you are requested to use Javadoc when documenting your code. Please do not submit the html files.

School solution

An executable version of the school solution can be invoked from the lab computers by typing: `~introcsp/bin/ex8/sequenceViewer fileName` , where fileName is a path to an input file.

Testing

It is recommended to check your JAR file by copying it to an empty directory, and running the automatic testers by executing the command: `~introcsp/bin/testers/ex8 ex8.jar` and verifying that all the tests pass successfully.
You may also download the tests JAR from here. Extract the contents of the JAR file using the shell command: `jar xvf ex8testing.jar` and follow the instructions in the file called **`TESTING.`**

Submission

Before you submit, remember to use constants where appropriate and refrain from duplicating code. Make sure to thoroughly test all the classes you implemented.

Don't forget to document your code (including javadoc) and follow all course coding style guidelines.
Submit a JAR file called **`ex8.jar`** containing **only** (“only”) the files:

- **`AdditionSequence.java`**
- **`ConsecutiveRatioSequence.java`**
- **`ConstantSequence.java`**
- **`CumulativeSumSequence.java`**
- **`DivisionSequence.java`**
- **`FibonacciSequence.java`**
- **`FiniteSequence.java`**
- **`IdentitySequence.java`**
- **`MultiplicationSequence.java`**
- **`PowerSequence.java`**
- **`SequenceFactory.java`**
- **`SequenceViewer.java`**
- **`README`**
- **`myInput.txt`**

Good luck!