CS2230 Computer Science II: Data Structures
Homework 5
Query processor using iterators

Due March 20 progress report, March 26 full submission
35 points

Goals for this assignment

- Learn how to write a variety of iterators using the Iterator<T> interface
- Learn how to use higher order functions in Java
- Learn how to use Java generic types
- Debug programs using JUnit tests

Submission Checklist

By March 20, 11:59pm you must fill in the progress report at https://uiowa.instructure.com/courses/78504/quizzes/96281 .

By March 26, 11:59 (or +slip days), You should have changes to the following files:

- Add1.java
- IntApply.java
- Apply.java
- FlatApply.java
- Filter.java
- Reduce.java
- TextQuery1b.java
- TextQuery2.java
- TextQuery3.java
- TextQuery4.java
- FlightsQuery.java

You will submit your files by **zipping the folder that contains your src/ and test/ folders**. If you followed Option#1 of getting_hw5 then this folder is called query-processor-master. If you followed Option#2 of getting_hw5 then this folder is called hw5 and is somewhere inside of NetBeansProjects.

Before you are done submitting, you must check the following.

- Do the tests pass?
- Does my zip file reflect **all the code I intend to turn in**? Download it from ICON and unzip to double check!

Prep materials: reading and lectures (about generic types, iterators, interfaces), lab7 on iterators, lab 8 on higher order functions, and quiz 5.

Description

In this project, you will build a Query Processor that can answer questions about data from Lists, text files, and CSV files (that is, spreadsheets). A "query" is sequence of Iterators chained together to process the input data.

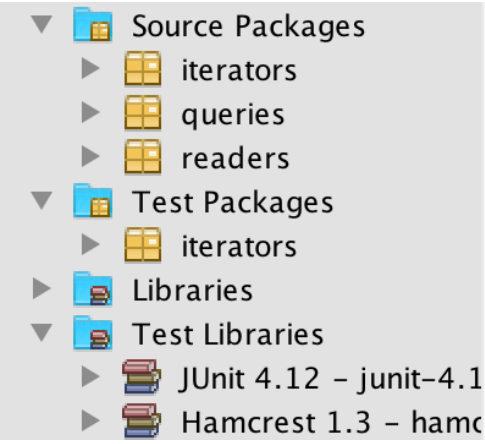
This project has a setup part and 7 parts. The best way to approach the project is in the order of these parts. You should pass all the tests specified in Part 1 before continuing to Part 2, and so forth.

A submission that passes all the tests in Part 3 and doesn't implement anything else will receive a higher grade than a submission that attempts all the parts and passes zero tests.

Part 0: Setup the project in NetBeans

1. Follow all of the directions in getting_hw5.pdf. **At this point, you can already submit the link to your repository to the ICON dropbox for HW5.**

When you are done with the directions, you should see something like the following in NetBeans:



2. Run some tests to make sure the project setup is working. Right click AddXTests.java | Run file (AddXTests.java is in Test Packages/iterators).

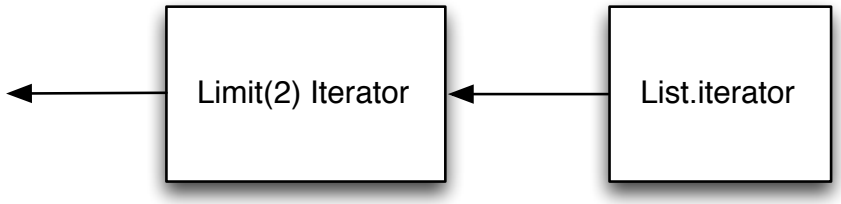
JUnit should finish running and report failed tests.

Background on Queries

A "query" is a question (or a transformation) on some input data. Input data might consist of a list of numbers, a list of strings, a text file, or a spreadsheet. The way we will *implement* a query is with a chain of Iterators, each one doing something to transform the input in a particular way.

Here is an example.
Let's suppose we have a list of numbers [10,50,1,400]. This list will be our input data.

A query might say "what are the first 2 elements of the input?". The chain of Iterators to answer this question would be



The Limit(2) Iterator reads one value at a time from List.iterator by calling the List.Iterator's next() method. Whoever wants to read the output of the query will call the next() method on the last Iterator in the chain; in this case, the last Iterator is Limit(2) Iterator.

Here is the sequence of next() and hasNext() calls that would occur to process this query.

Limit(2) Iterator	List.iterator
hasNext()...	
	hasNext() returns true
returns true	
next()...	
	next() returns 10
returns 10	
hasNext()...	
	hasNext() returns true
returns true	
next()...	
	next() returns 50

returns 50	
hasNext() returns false	

Notice that for Limit(2) Iterator to return a value from its next(), it must call its input Iterator's next() method. Also notice that the Iterators process just one element each time next() is called.

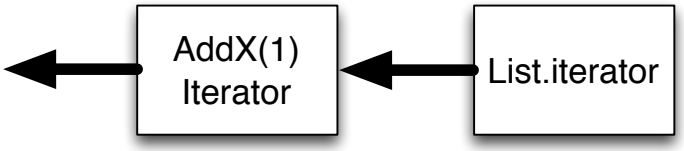
Run LimitTest.java. All the tests should pass without any changes (the above example can be found in moreTest). Take a look at Limit.java to see the implementation of the Limit Iterator. Answer the following question for yourself: How does Limit produce the next() element? **You will use Limit.java as an example to help you build additional Iterators** to run more interesting queries.

Part 1: AddX

End result of this part:

- pass all tests in these files
    - AddXTest.java

Here is a motivating query. Let's suppose we have a list of numbers [10,50,1,400]. This list will be our input data. The query is "add 1 to each element of the input". The chain of Iterators to answer this question would be



Here is the sequence of next() calls that would occur to process this query (we are omitting the hasNext() calls).

Add1Iterator	List.iterator
next()...	
	next() returns 10
returns 11	
next()...	
	next() returns 50
returns 51	
next()...	
	next() returns 1
returns 2	
next()...	
	next() returns 400

returns 401	

What you need to do

Fill in the implementation of the AddX class in AddX.java. We've already provided the fields and the constructor to get you started. You need to fill in the hasNext() and next() methods. **Keep in mind:** according to the Iterator interface, hasNext() may be called 0 or more times between every call to next(). You'll see that the tests check for this property.

Testing your code

Run the tests in AddXTest.java by right clicking that file and choosing "Run file". As in HW3, don't be alarmed by lots of failing tests. Start by trying to fix the simpler tests first, then move on to the more complicated ones.

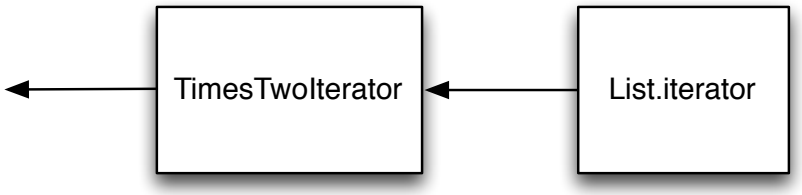
Remember, the code inside of one of these Test methods is not magical. It is creating an input list, building a query, then getting outputs from the query by calling next and hasNext.

Part 2: IntApply

End result of this part:

- pass all tests in these files
    - IntApplyTest.java

Here is a motivating query. Let's suppose we have a list of numbers [10,50,1,400]. This list will be our input data. The query is "multiply each element of the input by 2". The chain of Iterators to answer this question would be



The TimesTwoIterator reads one value at a time from List.iterator by calling the List.Iterator's next() method. Whoever wants to read the output of the query will call the next() method on the last Iterator in the chain, which is TimesTwoIterator.

Here is the sequence of next() calls that would occur to process this query (we are omitting the hasNext() calls).

TimesTwoIterator	List.iterator

next()...	
	next() returns 10
returns 20	
next()...	
	next() returns 50
returns 100	
next()...	
	next() returns 1
returns 2	
next()...	
	next() returns 400
returns 800	

Background: Higher Order Functions

To make the Iterators in this query engine more reusable, you are going to incorporate the idea of higher order functions. Functions that take arguments as functions or return functions are called *higher-order functions*. This sounds abstract, but it is very useful in practice as we will see in later questions.

Refer to Prelab8, Lab8, and lecture-022-hofs.pdf for examples of higher order functions and how to use them. And, here's another resource with some examples https://flyingbytes.github.io/programming/java8/functional/part1/2017/01/23/Java8-Part1.html.

The IntApply iterator

A common operation in query processing is to simply call some function, f, on the input element to get an output element. This operation is known as "Apply". In fact, the TimesTwoIterator above is a special case of Apply, where f just multiplies the input by two.

Here is our example query written in Java. We've used IntApply instead of TimesTwoIterator.

```
Integer[] inputValues = {10,50,1,400};
List<Integer> input = Arrays.asList(inputValues);
IntApply op = new IntApply(new TimesTwo(), input.iterator());
```

The IntApply constructor takes two arguments: the first is an object of type IntApplyFunction and the second is the input iterator.

IntApplyFunction is an interface with one method, `apply()`. Any class that "implements" IntApplyFunction must provide an implementation of `apply()`.

```
public interface IntApplyFunction {
            public int apply(int x);

}
```

In IntApplyTest.java you can see an example of how IntApply is used. We define a class TimesTwo that defines an apply() that multiplies its input by 2.

```
private class TimesTwo implements IntApplyFunction {

      @Override
      public int apply(int x) {
            return x*2;
      }
}
```

What you need to do

Fill in the implementation of the IntApply class in IntApply.java. You need to complete the constructor, hasNext, and next.

Testing your code

Run the tests in IntApply.java.

Part 3: Apply

End result of this part:

- pass all tests in these files
    - ApplyTest.java
- run the following Queries
    - TextQuery1a.java
    - TextQuery1b.java

The Apply iterator

Wouldn't it be nice if the IntApply operator worked for data that wasn't just integers? In fact, at the end of this part, you will run a query on some Text data. To get there, you will write a "generic" version of IntApply that can deal with any type of data.

Take a look at Apply.java. You'll see that it looks very similar to IntApply.java except that there are generic types InT and OutT. InT indicates the type of the input data. OutT indicates the type of the output data.

Notice that instead of an IntApplyFunction, Apply uses an ApplyFunction<InT, OutT>. This interface provides the generic apply() method that can take any input type and return any output type.

```
public interface ApplyFunction<InT, OutT> {
          public OutT apply(InT x);

}
```

Finally, when we want to use Apply, we will implement ApplyFunction. In ApplyTest.java you will see a few examples of generic apply functions. One of them is the TimesTwo class rewritten to implement ApplyFunction. Notice that both generic types are Integer to say that our multiply-by-2 operation takes an integer as input and produces an integer as output.

```
private class TimesTwo implements ApplyFunction<Integer, Integer> {

      @Override
      public Integer apply(Integer x) {
            return x*2;
      }
}
```

What you need to do

Fill in the implementation of the Apply class in Apply.java. We've already provided the fields and the constructor to get you started.

Testing your code

Run the tests in ApplyTest.java.

Try queries on real data!

Once you pass all those tests, try running TextQuery1a.java, which runs a query on text files in the provided sci.space/ folder. This data comes from a collection of newsgroup discussions from 1997[1].

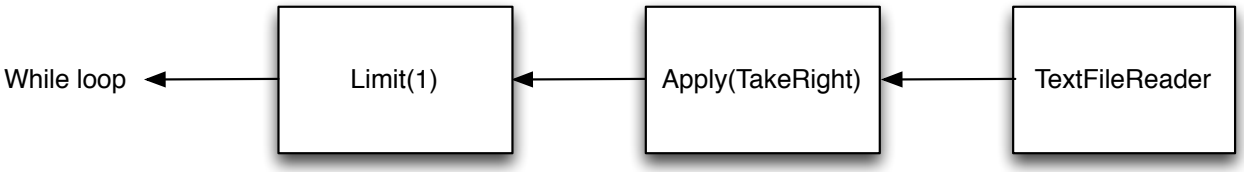
The query uses an iterator we've provided called TextFileReader (in src/readers) that reads all text files in the provided folder of text files. TextFileReader.next() returns objects of type

[1] http://qwone.com/~jason/20Newsgroups/

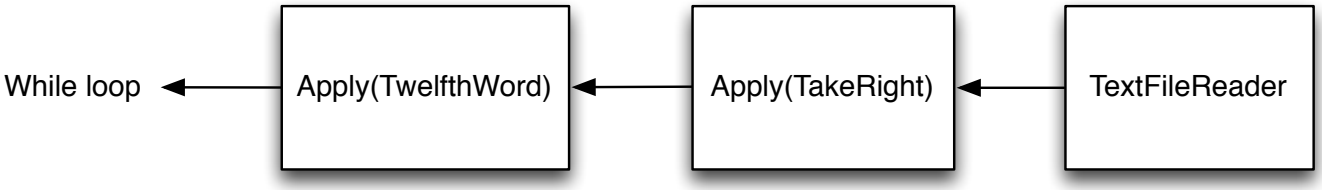
Pair<String,String>. The "left" element in the Pair is the filename and the "right" element in the Pair is the entire contents of that file. The default query just uses Apply to take the "right" element from a Pair, that is, return the file contents.

The while loop calls next() on the last Iterator and prints out the elements (i.e., the contents of *all* the text files).

Here is an illustration of the Iterators in TextQuery1a



Now, you need to implement a query yourself, in TextQuery1b.java. This query is very similar, except instead of the Limit, it takes the 12th word from every file. **All you need to do to finish the query is fill in the TwelfthWord class**. When you run the query, you'll see just the twelfth word from every file.



For your reference, you can find the expected output of the query in expected/TextQuery1b.txt.

Part 4: FlatApply

End result of this part:

- pass all tests in these files
  - FlatApplyTest.java
- run the following Queries
  - TextQuery2.java

Here is a motivating query. Let's suppose we have a list of numbers [10,50,1,400]. This list will be our input data. The query is "repeat each element 2 times".

This query will produce eight output elements: 10,10,50,50,1,1,400,400. At first glance, it seems like we may be able to accomplish this with Apply and a well chosen f. However, Apply only allows us to output exactly 1 element for each input element. We could certainly define an f whose return type is List<Integer>, but that would give us an output of 4 elements [10,10],[50,50],[1,1],[400,400], which is not quite what we want.

To write the query, we need a new Iterator called FlatApply. FlatApply is a generalization of Apply. It produces 0 or more output elements for each input element. This fact makes the implementation of FlatApply a bit more complicated than Apply. Since a single input may create many outputs, you'll need to keep a queue of pending elements that future calls to next() will return. When this queue becomes empty, you must fill it up by calling next() again on the input Iterator.

What you must do

Fill in the FlatApply class. Notice that its constructor takes a FlatApplyFunction. This interface has an apply method very similar to ApplyFunction, except it returns a List, which may have any number of elements in it.

IMPORTANT HINT: we suggest maintaining the following invariant in your FlatApply class.
Between calls to FlatApply.next() either:

a) *the queue is not empty*
b) *or, the queue is empty and input.hasNext() is false.*

By "between calls", we mean this invariant should always be true after the constructor finishes, except that it may be violated while executing the FlatApply.next() method.

Testing your code

Run FlatApplyTest.java. As before, try to work one at a time, starting with the simpler tests (emptyTest and oneTest).

Try queries on real data!

TextQuery2.java contains the query "return all the words longer than 24 characters". You must complete the query by implementing the LongerThan class (see code for details). **HINT:** the String.length method will be useful.

Part 5: Filter

End result of this part:

- pass all tests in these files
    - FilterTest.java

- run the following Queries
    - TextQuery3.java

Here is a motivating query. Let's suppose we have a list of numbers [10,50,1,400]. This list will be our input data. The query is "return the elements greater than 40". The output for this example would be the elements 50, 400.

As you saw in TextQuery2.java, we can use FlatApply to remove elements that don't pass a check (e.g., only return words longer than 12 characters). This removal of data based on a checking some property is commonly known as "filtering". Filtering is so common that it is worth implementing another Iterator called Filter.

Because Filter just a special case of FlatApply, we will use FlatApply to build Filter. In particular, we will use Java's concept of ***inheritance*** to borrow most of the functionality from the FlatApply class.

Take a look at Filter.java. You'll see that it "extends" the FlatApply class.

```
public class Filter<T> extends FlatApply<T,T> {
```

"Extends" is a lot like "implements", except that you use it on a class instead of an interface. The "extends" here means that that a Filter<T> *is a* FlatApply<T,T>. Therefore, a Filter<T> already has the useful next() and hasNext() methods defined by FlatApply.

NOTE: Why extend FlatApply<**T,T**>? Unlike FlatApply, Filter never changes the type of the data; it only keeps or removes each element. Therefore, the input type T is the same as the output type T.

What you need to do

Implement the Filter class. **You only need to fill in the inner class FilteringFlatApplyFunction.** You **do not** need to override the next() and hasNext() methods, and you **do not need to edit the constructor**.

The first argument to the Filter constructor is a Predicate<T>. The Predicate interface's one method, Predicate.check, is like ApplyFunction.apply, except it returns a *boolean* value. True indicates return the element; false indicates ignore the element.

Try queries on real data!

TextQuery3.java runs the query "return all filenames that contain the word 'Mars' " (now things are getting interesting!). Your job is to create a chain of Apply and Filter Iterators to implement this query.

Part 6: Reduce (you're almost there!)

End result of this part:

- pass tests in these files
    - Reduce.java
- run the following Queries
    - TextQuery4.java

Here is a motivating query. Let's suppose we have a list of numbers [10,50,1,400]. This list will be our input data. The query is "return the sum of the elements". The output of this query is always just one element long; for our example list, the output is 461.

We'll define an Iterator called Reduce that takes *all* the input and *combines* it to produce *one* output.

Reduce is a bit different from the other Iterators in the sense that it waits until it has seen *all* of the input before it returns an output. Also, a Reduce Iterator will return exactly one element before hasNext starts returning false.

Reduce depends on two functions defined by the ReduceFunction interface.

- `combine`: takes the accumulated value (soFar) and a new input (x) and returns a new accumulated value.
- `initialValue`: returns the *initial* accumulated value.

So for example, in the case of our sum example above, we would define a class that implements ReduceFunction, where the method `initialValue` returns 0 and method `combine` returns soFar+x.

NOTE: the type of soFar and x could be different. ReduceTest.java provides an example in the class MaxAsString.

Try queries on real data!

TextQuery4.java contains the query "count the number of occurrences of the word 'Mars' ". Borrow what you need from previous queries.

HINT: Reduce should be the last Iterator in this query. Create an inner class that implements ReduceFunction, defined to count inputs.

Part 7: Processing spreadsheets

End result of this part:

- run the following Queries
    - FlightsQuery.java

We have provided a comma separated values (CSV) file (flights-tiny.csv[2]) that contains spreadsheet data. Each line in the file represents one airline flight. A line contains several *fields*, which are integer and string values separated by commas. Each field is like a column in a spreadsheet.

The names of the fields in the csv are:

year	month	day of month	airline	flight number	origin city	dest city	cancelled	time

In FlightsQueries.java, you'll find a partially written query, where the Iterator called *records* returns elements which are FlightRecords, a simple object with the fields above.

Your job is to finish the query so that it computes "the number of flights that occurred in the year 2015".

Optionally run on more data

If you get the right answer on flights-tiny.csv, you can try running the query on flights-small.csv, Download flights-small.csv from https://uiowa.instructure.com/files/5755996/download?download_frd=1 . Put the file in the same directory where flights-tiny.csv is. Make sure to comment out the correct line in FlightsQuery.java:

```
//Iterator<String> lines = new LineFileReader("flights-small.csv"); // expects answer: 520718
Iterator<String> lines = new LineFileReader("flights-tiny.csv"); // expects answer: 5
```

Extra credit (up to 2 points)

You may only attempt this section, if you've finished the rest of the assignment.

These queries are challenging but should not require new Iterator classes.

- Turn in an additional file named **ExtraCreditFlightsQuery2.java** where the main method runs the following query.
    - Return the percentage (as a decimal number < 1) of flights between Los Angeles CA and New York NY that were cancelled.
- Turn in an additional file named **ExtraCreditFlightsQuery3.java** where the main method runs the following query.

[2] from Bureau of Transportation statistics https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time

    - Return the airline, origin city, dest city, and time (as a String[] with 3 elements) for the flight with the highest time.

Helpful tips

- You will need to define classes that implement the ApplyFunction, FlatApplyFunction, Predicate, or ReduceFunction interfaces in the queries. Make them inner classes within the same Java file and make them "static". See TextQuery1a.java for an example.
- Various problems related to Iterators
    - hasNext() crashing when called multiple times
    - calling hasNext() multiple times causes the Iterator to skip elements
    - next() crashing or returning bad values even when hasNext() returned true
    - hasNext() returns false too early / never returns false

  Try to identify an invariant for the Iterator you are working on. Then make sure that invariant holds before and after calls to hasNext() and next().

- Test xyz isn't passing! As in HW3, do some investigation to narrow down the source of the problem. Which line(s) of code are doing something different than you expected? Then, if you are still stuck, bring your question (and 1-3 hypotheses about what might be wrong) to a peer, Piazza discussion board, or a staff member in person.