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

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, using the Function<T,R> interface
- Learn how to use Java generic types
- Debug programs using JUnit tests

Prep materials: reading, knowledge checks, lectures about generic types, iterators, interfaces, prelabs/labs on interfaces, iterators

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.

Critical point #1: 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.

Critical point #2: You must make at least one git commit for every Part that you complete. This is at least 7 total commits for Parts 0-6. That means: before continuing from Part i to

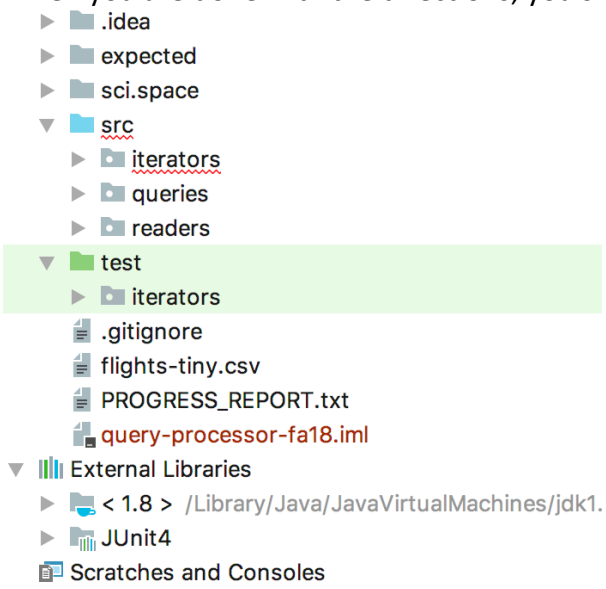
Part i+1, commit all your changes. Your commit history will show a gradual progress over time. Submissions with one giant commit for the whole assignment will earn 0 credit.

Your code must be at https://github.uiowa.edu/cs2230-sp19/hw5-hawkid to be submitted. There are no exceptions, so get help early if you are having trouble with Git.

Part 0: Setup the project in IntelliJ

1. Follow all of the directions in getting_hw5.pdf.

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



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).

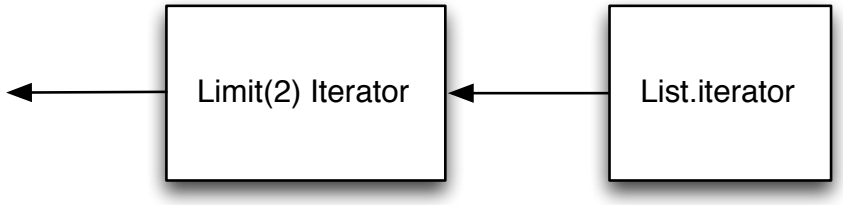
JUnit should finish running and report failed tests.

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.

Leave comments in Limit.java to describe how it works. You will write comments at:

- Each instance variable

- Inside the constructor
- Inside of hasNext()
- Inside of next()

In the rest of this assignment, you will use Limit.java as an example to help you build additional Iterators to run interesting queries.

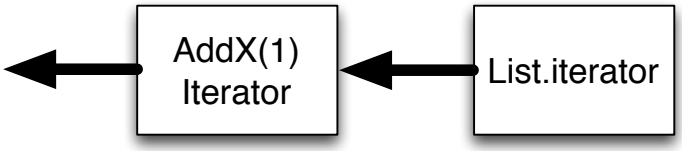
STOP: Before you continue to the next Part, ***commit*** your code in Github Desktop with a message like "Finished Part 0". To share your progress with the course staff, you should also now ***push*** to your repository
https://github.uiowa.edu/cs2230-sp19/hw5-hawkid
And verify that your latest commit is in there by going to that URL in your web browser. **If you cannot get this to work**, get help as soon as possible.

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.

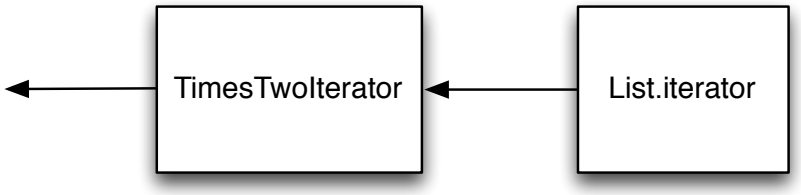
STOP: Before you continue to the next Part, ***commit*** your code in Github Desktop with a message like “Finished Part 1”. To share your progress with the course staff, you should also now ***push*** to your repository
https://github.uiowa.edu/cs2230-sp19/hw5-hawkid
And verify that your latest commit is in there by going to that URL in your web browser.

Part 2: IntTransform

End result of this part:

- pass all tests in these files
    - IntTransformTest.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 course materials for examples of higher order functions (abbreviated sometimes as "HOFs") 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 IntTransform 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 "Transform". In fact, the TimesTwoIterator above is a special case of Transform, where f just multiplies the input by two.

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

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

The IntTransform constructor takes two arguments: the first is an object of type Function<Integer, Integer> and the second is the input iterator.

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

https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

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

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

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

What you need to do

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

Testing your code

Run the tests in IntTransform.java.

STOP: Before you continue to the next Part, ***commit*** your code in Github Desktop with a message like “Finished Part 2”. To share your progress with the course staff, you should also now ***push*** to your repository

https://github.uiowa.edu/cs2230-sp19/hw5-hawkid
And verify that your latest commit is in there by going to that URL in your web browser.

Part 3: Transform

End result of this part:

- pass all tests in these files
  - TransformTest.java
- run the following Queries
  - TextQuery1a.java
  - TextQuery1b.java
  - TextQuery1c.java

The Transform iterator

Wouldn't it be nice if the IntTransform 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 IntTransform that can deal with any type of data.

Take a look at Transform.java. You'll see that it looks very similar to IntTransform.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 using a Function<Integer, Integer>, Transform has two generic types InT and OutT and uses a Function<InT, OutT>.

Finally, when we want to use Transform, we will implement Function. In TransformTest.java you will see a few examples of generic apply functions. One of them is the TimesTwo class rewritten to implement Function. 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 Function<Integer, Integer> {

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

What you need to do

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

Testing your code

Run the tests in TransformTest.java.

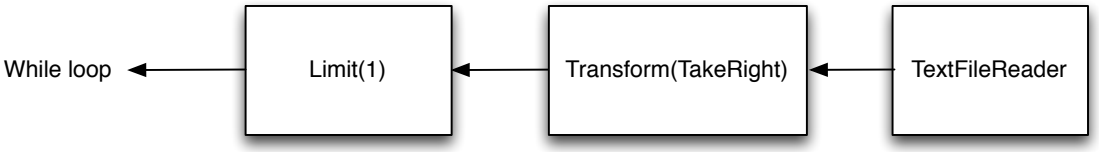
Try queries on real data!

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].

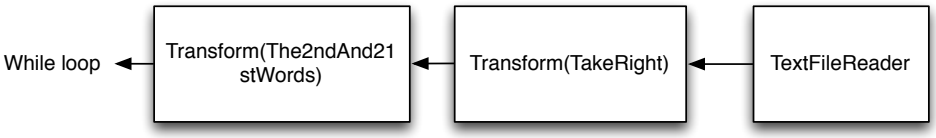
The query uses an iterator we've provided called TextFileReader (in src/readers) that reads all text files in the provided folder of text files. TextFileReader.next() returns objects of type Pair<String,String>. The "left" element in the Pair is the filename and the "right" element in the Pair is the entire contents of that file. The default query just uses Transform 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 2nd and 21st words from every file. **All you need to do to finish the query is fill in the The2ndAnd21stWords class**. When you run the query, you'll see just those two words, one line per input file.



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

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

Next, finish the query in queries/TextQuery1c.java. The query finds all the words that start with Z or z. You can find the expected output in expected/TextQuery1c.txt.

STOP: Before you continue to the next Part, ***commit*** your code in Github Desktop with a message like "Finished Part 3". To share your progress with the course staff, you should also now ***push*** to your repository
https://github.uiowa.edu/cs2230-sp19/hw5-hawkid
And verify that your latest commit is in there by going to that URL in your web browser.

Part 4: TransformToMany

End result of this part:

- pass all tests in these files
  - TransformToManyTest.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 Transform and a well chosen f. However, Transform 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 TransformToMany. TransformToMany is a generalization of Transform. It produces 0 or more output elements for each input element. This fact makes the implementation of TransformToMany a bit more complicated than Transform. 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 TransformToMany class. Notice that its constructor takes a Function<InT, List<OutT>>. This means that apply returns a List, which may have any number of elements in it.

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

the queue is not empty || the queue is empty and !input.hasNext()

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

Testing your code

Run TransformToManyTest.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.

STOP: Before you continue to the next Part, ***commit*** your code in Github Desktop with a message like “Finished Part 4”. To share your progress with the course staff, you should also now ***push*** to your repository
https://github.uiowa.edu/cs2230-sp19/hw5-hawkid
And verify that your latest commit is in there by going to that URL in your web browser.

Part 5: SuchThat

End result of this part:

- pass all tests in these files
  - SuchThatTest.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 TransformToMany to remove elements that don't pass a check (e.g., only return words longer than 24 characters). This removal of data based on

a checking some property is so common that it is worth implementing another Iterator called SuchThat.

What you need to do

Implement the SuchThat class. Make sure you **uncomment** the "implements Iterator" before you start.

The first argument to the SuchThat constructor is a Function<InT, Boolean>. Notice that the return type of the apply method is hard-coded to Boolean (i.e., the type whose value can be true or false). True indicates return the element; false indicates ignore the element.

Optional hint: feel free to use TransformToMany in your implementation of SuchThat. It is recommended although not required.

Try queries on real data!

TextQuery3.java runs the query "return all filenames that contain the word 'Mars' *or* the word 'alien'" (now things are getting interesting!). Your job is to create a chain of Transform and SuchThat Iterators to implement this query. Find the expected output in TextQuery3.txt.

STOP: Before you continue to the next Part, ***commit*** your code in Github Desktop with a message like "Finished Part 5". To share your progress with the course staff, you should also now ***push*** to your repository
https://github.uiowa.edu/cs2230-sp19/hw5-hawkid
And verify that your latest commit is in there by going to that URL in your web browser.

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 a single element; for the example list [10,50,1,400], the output is 461.

We'll define an Iterator called Reduce that takes *all* the input elements and *combines* them 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 is also different in that its constructor takes a BiFunction<OutT, InT, OutT>[2] instead of a Function. This interface represents a function that takes *two inputs*, of types OutT and InT, and produces one output of type OutT.

- `apply`: takes the accumulated value (soFar) and a new input (x) and returns a new accumulated value.

Reduce's constructor also takes an initial value. This provides the Reduce iterator with its first value for sofar.

So for example, in the case of our sum example above, we would define a class that implements BiFunction<OutT, InT, OutT>, where the method `apply` returns soFar+x. And, we'd call the constructor with initialValue=0.

NOTE: the type of soFar and x could be different. ReduceTest.java provides two examples of BiFunction<OutT, InT, OutT>, one where the types differ (MaxAsString) and one where the types are the same (MinInteger).

Try queries on real data!

TextQuery4.java contains the query "count the number of occurrences of the words 'Mars' or 'alien'". Borrow what you need from previous queries. Find the expected output in TextQuery4.txt.

HINT: Reduce should be the last Iterator in this query. Create an inner class that implements BiFunction<OutT, InT, OutT>.

HINT: Counting and summing all input elements can both be done using Reduce. But, the BiFunction for these two operations is a bit different.

STOP: Before you continue to the next Part, ***commit*** your code in Github Desktop with a message like "Finished Part 6". To share your progress with the course staff, you should also now ***push*** to your repository

[2] https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html

https://github.uiowa.edu/cs2230-sp19/hw5-hawkid
And verify that your latest commit is in there by going to that URL in your web browser.

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[3]) 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 the "Homework 5" Assignment on ICON. 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
```

STOP: To **submit** your finished assignment, ***commit*** your code in Github Desktop with a message like "Finished Part 7". Then also now ***push*** to your repository
https://github.uiowa.edu/cs2230-sp19/hw5-hawkid
And verify that your latest commit is in there by going to that URL in your web browser.

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

Helpful tips

- You will need to define classes that implement the Function and BiFunction interfaces in the queries. Make them inner classes within the same Java file and make them "static". **See TextQuery1a.java for an example.** That file uses the Function Pair.Right(), defined in Pair.java.
- 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

    The easy way to implement an iterator: create a list of elements in the constructor.

    The more interesting way to implement an iterator: 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.

Extra credit (up to 2 points)

You may only attempt this section if you've finished the rest of the assignment. **Make sure to commit all previous work before starting any extra credit changes.**

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 between 0 and 100) of flights between “Chicago IL” and “Des Moines IA” that were cancelled.
- Turn in an additional file named **ExtraCreditFlightsQuery3.java** where the main method runs the following query.
    - Return the airline, origin city, dest city, and time (as a String[] with 4 elements) for the flight with the lowest time.

You must have **one commit for each of the two queries** that **only includes the changes from the extra credit**, with the description: “Extra Credit #1 More Flights”.

Extra credit (up to 2 points)

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

Write an interesting query on 1 of the 2 CSV files from Application Activity 2: Lists. To read the CSV file, you can either use FlightsQuery as an example or you can use the CSV library used in the Application Activity. If you use the CSV library, you should write a new reader and put it in the readers/ folder.

- Put the query, itself, in the queries/ folder and call it CensusQuery.java.
- You must also write the query in English as a comment in your java file.
- What is an interesting query?
  - The query should have a small answer that a human could understand. That means that it will require at least one SuchThat, TransformToMany, or Reduce iterator, not just Transform.

You must have **one commit** that **only includes the changes from the extra credit**, with the description: “Extra Credit #2 App Activity”.

Extra credit (up to 2 points)

As of Java 8, Java supports *lambda expressions*, which are unnamed functions that can be used anywhere that a Function or BiFunction is expected.

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Replace ***all*** inner classes that implement Function or BiFunction in HW5 with a lambda expression.

You must have **one commit** that **only includes the changes from the extra credit**, with the description: “Extra Credit #3 Lambdas”.

Image credits

stop sign by Libby Ventura from the Noun Project