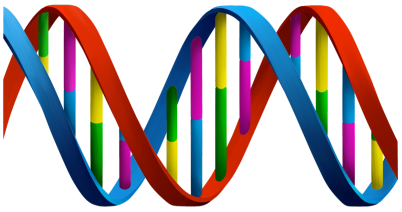
CS 46B
Homework 4
Curing cancer, 1 line of code at a time
Due 11:59 PM, October 19, 2016



The major hope of modern medicine is to develop personalized treatment for cancer patients. This is already happening for a few kinds of cancer, with great success.

Every minute, around 50 million of our cells die and are replaced. The replacement cells come from healthy cells that divide into 2 identical copies that then grow to full size. Actually, the copies aren't perfectly identical. The DNA in the original cell contains 3 billion bases. It's impossible to perfectly replicate 3 billion of *anything*, and get it right every time. Sometimes there's an error, so that for example a thymine ("T") appears where a cytosine ("C") should be. Usually these errors, which are called *mutations*, do no harm, but sometimes they disrupt a part of the DNA that protects the cell against becoming a cancer cell. Then the cell divides, becoming 2, then 4, then 8, then 2^n neighboring cancerous cells: a tumor.

There are many different kinds of cancerous mutation; often these different mutations respond to different kinds of treatment. Knowing the DNA sequence of the tumor cells gives doctors a huge advantage in prescribing therapy.

As you saw in lecture, patient care can begin with extracting DNA from a tumor, and determining its sequence using a machine that outputs a text file in "fastq" format. Unfortunately it is not yet possible to determine the ACGT…

sequence of entire DNA molecules. The machine chops the molecule into lots of segments of a few hundred bases; these segments are called *reads*. We wish the fastq file could contain a single record with a very long sequence. Instead, until some hard technical problems are solved, fastq files contain thousands to millions of relatively short records.

The fastq file format, as you saw in lecture on Feb. 24, has one 4-line record for each read. The lines are:

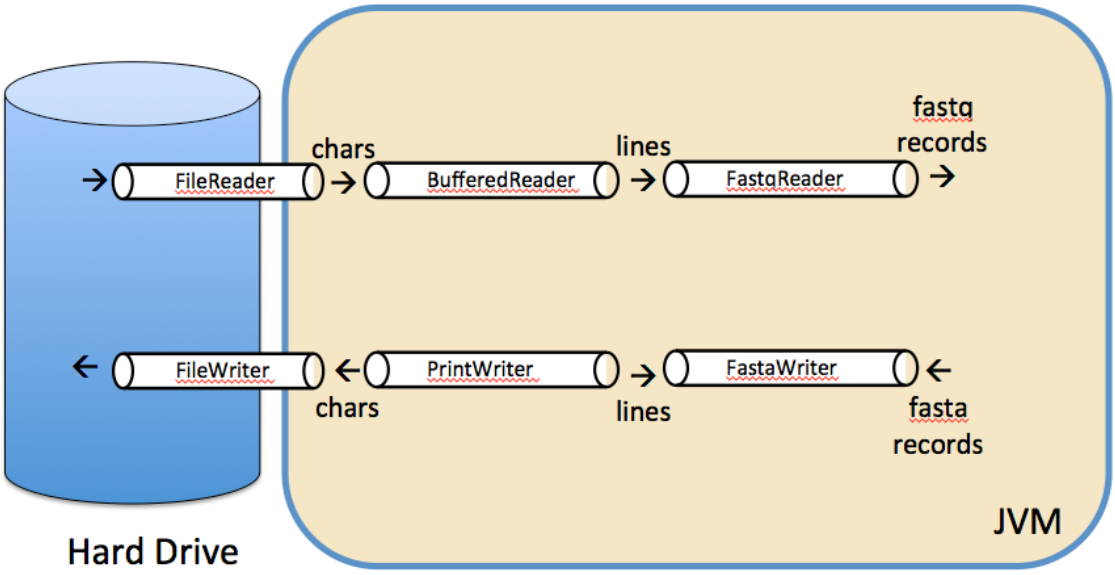
1) The "defline": starts with @, followed by a unique identifier. All records in the fastq file are supposed to have different deflines. On rare occasions, a bug in the sequencer causes 2 or more records to have the same defline.
2) The sequence: a string of (usually) several hundred characters that represent the DNA sequence of the reads. The only legal characters (for this homework) are A, C, G, and T.
3) + (just a plus sign all alone on a line).
4) The quality. This is a string, exactly the same length as the sequence, of mysterious chars that encode the sequencing machine's confidence that the corresponding char in the sequence is correct. The char representing maximum confidence is the exclamation mark (!).

You also saw that for bioinformatic analysis, fastq files are commonly converted into fasta format. A fasta file also has 1 record per read, but there are differences:

- Fasta files have only 2 lines per record: A defline and a sequence line.
- The defline starts with > rather than @.

For this assignment you will write a Java app that reads a fastq file and creates a fasta file. Records in the fastq *might or might not* meet some quality threshold, and *might or might not* have unique deflines. Records in the fasta *will* all meet or exceed a quality threshold, and *will* all have unique deflines.

In class, you saw a method that almost does that, while reading from a BufferedReader and writing to a PrintWriter. For this assignment, you will use a different approach. You will create classes FastqReader (which will read from a BufferedReader) and FastaWriter (which will write to a PrintWriter).



In Eclipse, create a new package called dna, and import the 8 starter files that you downloaded with this assignment

1) DNARecord.java
2) FastqRecord.java
3) FastaRecord.java
4) FastqException.java
5) FastqReader.java
6) FastaWriter.java
7) FileConverter.java
8) DNAGrader

Then complete the starter files as described below.

The FastqRecord class

This class should implement DNARecord and Comparable<FastqRecord>, and should have:

- 3 String instance variables: defline, sequence, and quality.
- A constructor that initializes the instance variables. If the defline does not start with the correct character, the ctor should throw FastqException with a helpful message. Yes, ctors can throw exceptions just like methods; be sure to add "throws FastqException" to the ctor

declaration. Here are some possible messages, in increasing order of helpfulness:
    - An empty or null string
    - Zzup yo?
    - Oops
    - Bad fastq record
    - Bad defline in fastq record
    - Bad 1st char in defline in fastq record
    - Bad 1st char in defline in fastq record: saw X, expected @
- Methods that satisfy the interface by returning the defline and the sequence.
- An equals() method that checks for deep equality of all 3 instance variables.
- A compareTo() method that first compares deflines, then (if necessary) sequences, then (if necessary) qualities.
- A boolean qualityIsHigh() method that returns true if and only if the first quality char is an exclamation mark (!). Of course in real life this method would be a lot more complicated, and might need a lot of helper methods.
- A toString() method that returns the defline, then a newline char, then the sequence, then a newline char, then a +, then a newline char, then the quality, then a newline char. (The newline char is \n).
- A hashCode() method that returns the sum of the hash codes of defline, sequence, and quality.

The FastaRecord class

This class should implement DNARecord and Comparable<FastaRecord>, and should have:

- 2 String instance variables: defline and sequence.
- A constructor that takes 2 args – the defline and the sequence – and initializes the instance variables. As with FastqRecord, do a precondition check to make sure the defline starts with the correct character (it's '>' for fasta records).
- Another ctor with 1 arg – a FastqRecord – that initializes the instances variables with values from the FastqRecord.

- Methods that satisfy the interface by returning the defline and the sequence.
- An equals() method that checks for deep equality of the 2 instance variables.
- A compareTo() method that first compares deflines, then (if necessary) sequences.
- A toString() method that returns the defline, then a newline char, then the sequence, then a newline char.
- A hashCode() method that returns the sum of the hash codes of defline, and sequence.

The FastqException class

This class should extend Exception (*not* RuntimeException, because we want it to be checked). Provide one constructor whose arg is a String. Pass the String to the superclass constructor that takes a single String arg.

The FastqReader class

FastqReader should not extend any superclasses or implement any interfaces. It should have one instance variable: a BufferedReader named theBufferedReader.

This class should provide a single-arg ctor that initializes theBufferedReader from its arg.

The class should also have the following method:
public FastqRecord readRecord() throws IOException, FastqException
This method should read a line from the buffered reader. If that line is null, the input file is at the end, and the method should return null. Otherwise the method should read 3 more lines and return a FastqRecord.

The FastaWriter class

FastaWriter should not extend any superclasses or implement any interfaces. It should have one instance variable: a PrintWriter named thePrintWriter.

This class should provide a single-arg ctor that initializes thePrintWriter from its arg. The class should also have the following method:

```
public void writeRecord(FastaRecord rec) throws IOException
```

This method should write the fasta record, in correct fasta format, to thePrintWriter.

The FileConverter class

This class should have 2 instance variables of type File, named fastq and fasta. Provide a ctor that has 2 File args and initializes the instance variables.

The class should also have a convert() method and a main() method.

The convert() method should declare that it throws IOException. Any other exception types thrown in the body of convert() should be caught and handled inside convert(). The method should

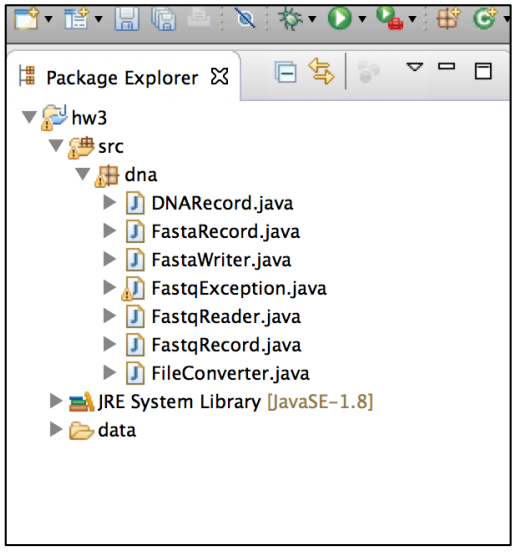
1) Create a FastqReader that reads from the fastq file specified by the fastq instance variable.
2) Create a FastaWriter that writes to the fasta file specified by the fasta instance variable.
3) Read each fastq record until the end of the fastq file is reached. Do nothing with any invalid records (i.e. records where the defline didn't start with @). Do nothing with valid records where the defline has already appeared in an earlier record. For valid records where the defline has not yet appeared in an earlier record, create a fasta record and write it using the FastaWriter.
4) Close all readers and writers that have close() methods, in reverse order of creation.

The main() method is provided for you. It reads and converts the fastq file that you downloaded with this assignment. The next section tells you what to do with the fastq file.

The Input File, and Testing

Notice that the main() method reads a fastq file in a directory called data, and writes a fasta in the same directory. You will need to create this directory in Eclipse, and import HW4.fastq into it. You will also need to enable assertions in Eclipse.

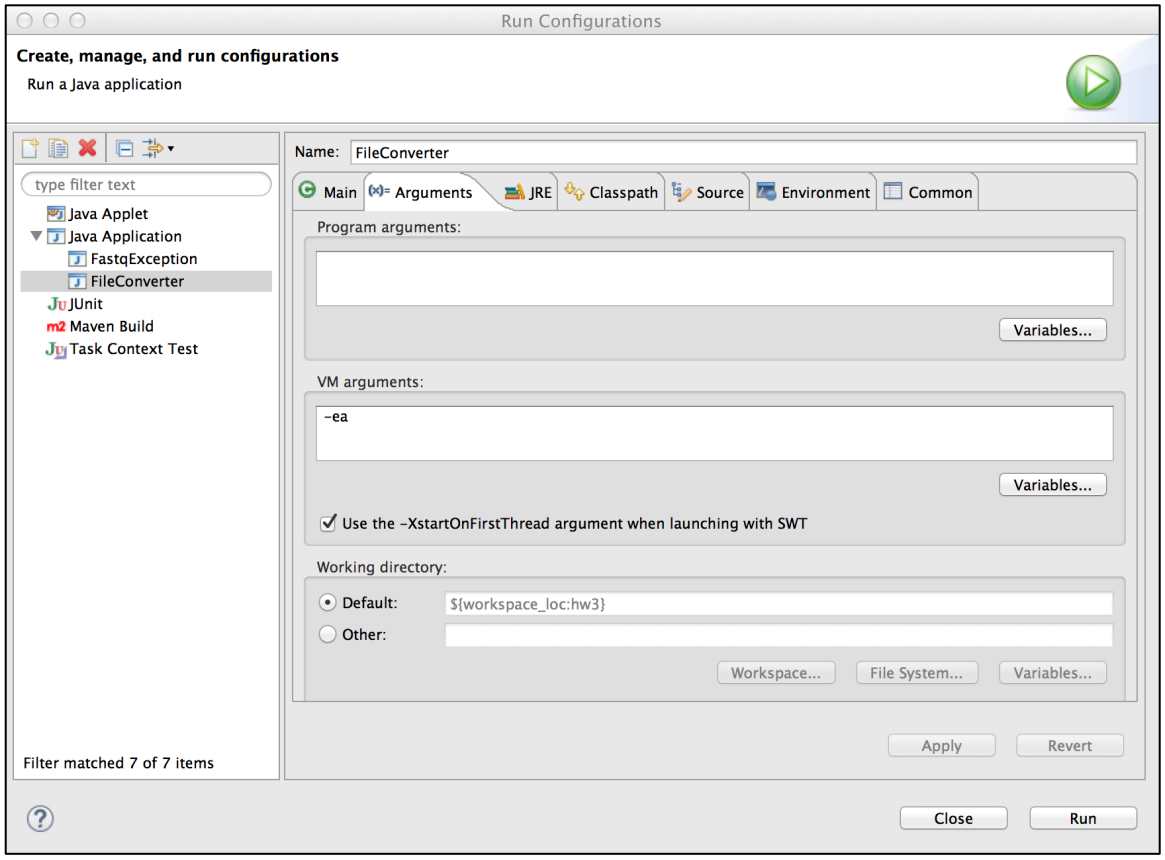
To create the data directory, right-click on your project name in the package explorer ("hw3" in the figure below … it's from last semester) and select New -> Folder in the popup menu. When prompted for the folder name, enter "data". You should see the new directory in the Package Explorer, at the same level as src. If it isn't at the right level, delete it and start again; it has to be in the right place for main() and the autograder to find it.



To import the fastq file, drag the icon for HW4.fastq into the icon of the data folder. A dialog box will ask you if you want to copy files or link to files; choose "copy files". Open the data folder by clicking on its triangle; you should see HW4.fastq.

To enable assertions, select FileConverter in the Package Explorer. Go to the main Eclipse menu and select Run -> Run Configurations… Then click on the Arguments tab. Be sure that FileConverter is selected in the list on the left. If it isn't selected, select it there. Type –ea into the VM Arguments field (*not* the Program Arguments field) as shown below, and click Apply. If the assertion is

raised, you probably created your data folder in the wrong place, or forgot to import HW4.fastq.



Now whenever you run the FileConverter app in Eclipse, assertions will be enabled. If you didn't import HW4.fastq correctly, the assertion message might be helpful.

Run the app. If it runs correctly, it will create a file called HW4.fasta in the data folder. Unfortunately, when you do this the first time, you won't see HW4.fasta in the data folder. Eclipse doesn't know when an app writes a new data file. Right-click on the data icon and select Refresh in the popup menu. Now if you don't see HW4.fasta it's because something went wrong in your program.

Check your work. Double-click on the fastq file to open it. Look at the records and decide which ones are high-quality and valid. Then open the fasta file,

and verify that it only contains fasta versions of the high-quality valid records.

Run the DNAGrader app to see what your grade will be.

Submitting

As usual, export your project and upload. Use the jar command on the command line to be sure that your jar contains all your .java files. Don't submit HW4.fastq or HW4.fasta.

Comments

Put a comment at the beginning of each class; at the beginning of most methods, and within methods if the method proceeds in several steps. For example, your convert() method will build its input stream, build its output stream, do the work, and then close its resources.

The Last Paragraph

This is the biggest assignment so far in 46B, and your code probably won't work right the first time. That's the way things are with realistic-size programs. Get comfortable with the process of figuring out what results you should see, and why you don't see them. START BY THINKING, NOT BY ASKING. You are here to develop your own skills at solving this kind of problem; debugging is part of the process, so you might as well enjoy it! (It's a lot less fun if you waited until the last minute to start the assignment.) Think about ways to insert temporary println statements, or use the debugger, or write a main() method that tests behavior of methods by passing very simple inputs. Or use "assert". For example, you could test FastqReader by creating a fastq file that contains one obviously correct record with a very short sequence, and putting println lines in readRecord() (or stepping through it with the debugger) to make sure the right thing happens. Then you can make the record invalid by changing @ to something else, and again seeing what happens.