CSE 142: Computer Programming I Spring 2021

Take-home Assessment 6: YazInterpreter *due May 18, 2021, 11:59pm*

This assignment will assess your mastery of the following objectives:

- Write a functionally correct Java program to produce specified console and file output.
- Use `Scanner` and `File` to read input from a file.
- Use `PrintStream` and `File` to write output to a file.
- Use methods of the `String` class to process and manipulate string values.
- Use methods to manage information flow and add structure to programs.
- Follow prescribed conventions for spacing, indentation, naming methods, and header comments.

Background

Note: You do not need to read this section to complete the assessment, but it provides some helpful context that may make the assessment easier to understand.

Throughout the quarter, we have been working with the programming language Java. Java is an example of a **compiled language**, meaning that before we can run our code, we need to run it through a tool called a *compiler* to translate it into a language that the computer itself can understand and execute. But not all languages work this way. Some languages are what are called **interpreted languages**, meaning that the source code in the language can be read and executed directly using a tool called an *interpreter*. The language you will work with on this assessment, YazLang, is an example of an interpreted language.

Sample Execution

```
Welcome to YazInterpreter!
You may interpret a YazLang program and output
the results to a file or view a previously
interpreted YazLang program.

(I)nterpret YazLang program, (V)iew output, (Q)uit? I
Input file name: input.txt
File not found. Try again: yazlang.txt
File not found. Try again: interpret-file.txt
Output file name: interpret-out.txt
YazLang interpreted and output to a file!

(I)nterpret YazLang program, (V)iew output, (Q)uit? View
(I)nterpret YazLang program, (V)iew output, (Q)uit? vi
(I)nterpret YazLang program, (V)iew output, (Q)uit? v
Input file name: interpret-out.txt

-9 -6 -3 0 3 6
39F
gucci ganggucci ganggucci ganggucci ganggucci ganggucci ganggucci gang
11C
humuhumunukunukuapua'a
5 12 19 26 33
24F

(I)nterpret YazLang program, (V)iew output, (Q)uit? q
```

Sample Execution 2

```
Welcome to YazInterpreter!
You may interpret a YazLang program and output
the results to a file or view a previously
interpreted YazLang program.

(I)nterpret YazLang program, (V)iew output, (Q)uit? V
Input file name: simple-out.txt

15C
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
abba

(I)nterpret YazLang program, (V)iew output, (Q)uit? interpret
(I)nterpret YazLang program, (V)iew output, (Q)uit? I
Input file name: simple.txt
Output file name: blah output.txt
YazLang interpreted and output to a file!

(I)nterpret YazLang program, (V)iew output, (Q)uit? q
```

Program Behavior

In this assessment, you will create an interpreter for the programming language YazLang. (This language was named a former CSE 142 Head TA and Instructor, Ayaz, who led development of the language and this assignment.) When interpreting a YazLang file, the program prompts the user for input and output file names (user input is underlined in the Sample Execution examples above). Then the program reads and executes the YazLang commmands in the input file and outputs the results to a different file. The user can later view the output file that was created or quit the program.

Console Output and Viewing a File

File Prompting

Depending on the command that the user selects, they will be asked to provide an input and/or output file name. Note that **all console input should be read in using the `Scanner`'s `nextLine()` method.**

- **Input Files:** When prompting for an input file, the user may respond with the name of an input file that does not exist. If the file does not exist, the user should be reprompted until they enter the name of a file that does exist. See the logs of Sample Execution on the first and second pages for examples of this reprompting behavior.

- **Output Files:** When prompting for an output file, the user may similarly respond with the name of a file that does not exist. In this case, you should rely on the default behavior of the input/output approaches we use. By default, if the output file does not exist, a new file will be created. And if the file does already exist, its contents will be overridden.

You should assume that the input and output files are not the same.

Menu

The program's menu should work properly regardless of the order or number of times its commands are chosen. For example, the user should be able to run each command (such as `I` or `V`) many times if desired. The user should also be able to run the program again and choose the `V` option without first choosing the `I` option on that run, or to run the program and immediately quit with the `Q` option if so desired. Menu options should be case-insensitive (e.g. both `Q` and `q` should cause the program to quit). If an invalid option (anything other than `I`, `V`, or `Q` in any casing) is entered, the user should be reprompted.

Viewing Output

When the user enters V from the menu, they should then be prompted to enter an input file to view. If the input file does not exist, the user should be reprompted until they enter the name of a file that does exist (see the **"File Prompting"** section for more details).

When you are viewing a given input file, you are simply reading and printing its contents, unchanged, to the console. This functionality *can* be used to view previously-interpreted YazLang files, but it can be used on any given input file that exists. Therefore, you do not need to test that the specified file is a YazLang output file. Just output the file's contents (even if it is a different type of file).

Interpreting and File Output

Interpreting YazLang Files

Sample Input File (interpret.txt)

```
RANGE -9 9 3
CONVERT 4 C
REPEAT "gucci_gang" 7
CONVERT 53 F
REPEAT "humu" 2 "nuku" 2 "apua'a" 1
RANGE 5 35 7
CONVERT -4 C
```

When the user enters I from the menu, they should then be prompted to enter an input file and an output file. The input file should contain YazLang commands. Your program should then read the input file, execute each command, and print the output to the output file. If the input file does not exist, the user should be reprompted until they enter the name of a file that does exist, but no reprompting is necessary for the output file (see the **"File Prompting"** section for more details).

Sample Output File (interpret-out.txt)

```
-9 -6 -3 0 3 6
39F
gucci ganggucci ganggucci ganggucci ganggucci ganggucci ganggucci gang
11C
humuhumunukunukuapua'a
5 12 19 26 33
24F
```

YazLang Commands

YazLang consists of three commands: CONVERT, RANGE, and REPEAT. These commands are described in the table on the next page.

The syntax for YazLang is much simpler (and more limited) than Java's. Every YazLang command follows this pattern:

$$\texttt{COMMAND}\ arg_1\ arg_2\ \ldots\ arg_n$$

The three commands will always appear exactly as CONVERT, RANGE and REPEAT (all uppercase) with the appropriate arguments

That is, every command consists of a single token indicating the command to be executed, followed by some number of arguments. Some commands take a specific number of arguments, while others may take any number of arguments. Some commands may also take no arguments, in which case the command token itself is considered a complete command. There will be one or more spaces or tabs between the command and the arguments, and between each argument. In a YazLang program file, each command is on its own line.

There are several example input and output files on the course website, along with supplemental videos explaining the program's desired behavior. **We strongly recommend looking over these additional resources to aid your understanding of how the program is intended to function before beginning your implementation.**

Command	Arguments	Description	Examples	Example Output

`CONVERT`	Always takes exactly two arguments:
	• arg_1: the temperature to convert, as an integer.
	• arg_2: either `C` or `F`, indicating arg_1's units of the temperature.
	arg_2 will always be either `C` or `F` (case-insensitive).
		Converts a temperature from Celsius to Fahrenheit or vice versa using the following formulas:
		$$F = 1.8 * C + 32$$
		$$C = (F - 32)/1.8$$
		If the temperature is currently in Celsius (that is, arg_2 is `C`), it should be converted to Fahrenheit. If the temperature is currently in Fahrenheit, it should be converted to Celsius. The output should be given as an integer, with any decimal places truncated (which can be achieved by casting the resulting calculation to an `int`), and should indicate the new units.
			`CONVERT 0 c`
			`CONVERT 32 F`
			`CONVERT 9 C`
			`CONVERT 9 f`
				`32F`
				`0C`
				`48F`
				`-12C`

`RANGE`	Always takes exactly three arguments:
	• arg_1: the first integer to be printed.
	• arg_2: the first integer to not be printed.
	• arg_3: the amount to increment by.
	arg_3 will always be greater than zero.
		Prints a sequence of integers starting from arg_1 and incrementing by arg_3 until a value greater than or equal to arg_2 is reached. Does not print arg_2 or any value greater than it. Produces no output if arg_1 is greater than or equal to arg_2.	`RANGE 0 5 1`
			`RANGE 1 10 9`
			`RANGE 2 1 1`
				`0 1 2 3 4`
				`1`

`REPEAT`	Takes an arbitrary number of arguments, alternating between strings and integers. The number of arguments will always be even (but might be zero). String arguments will be enclosed in quotation marks, and may contain underscores. Integer arguments will be greater than or equal to zero.	Prints out each string argument repeated the number of times indicated by the following integer argument. The string arguments should have the outer quotation marks removed and underscores replaced with spaces before printing.	`REPEAT "a" 5 "B" 2`
			`REPEAT "yo_yo" 1 "_"a" 1`
			`REPEAT "a" 1 "b" 0 "c" 2`
			`REPEAT`
				`aaaaaBB`
				`yo yo "a`
				`acc`

Creative Aspect (my-command.txt)

There are only three commands in YazLang at the moment, and to come up with more we have decided to crowdsource! Along with your program, submit a file called `my-command.txt` with a proposal for a new command to add to YazLang. Your proposal must include the following elements:

- The name of the command
- The arguments the command will take
- A description of what the command does
- At least one sample input and sample output

You should format your proposal like the example below. We have also posted examples on the course website. **You do *not* need to provide an implementation for your custom command.**

`repeat-proposal.txt`

```
REPEAT

REPEAT takes an arbitrary number of pairs of Strings and integers and
creates one large string with each string repeated the number of times
indicated by the following integer.

Input: REPEAT "ha" 3 "_" 1 "lol" 2
Output: "hahaha lollol"
```

Development Strategy

Once again, this assessment will be best approached in smaller chunks. We recommend the following strategy:

"Hard-coding" refers to embedding a value directly in your program rather than accepting it as input. For example, you might start by having your program always process the command `CONVERT 0 C` instead of a different command each time it runs.

(1) **File prompting:** Add code to prompt the user for an input to use (be sure to remove your hard-coding when you reach this point).

(2) **File reprompting:** Handle the case when the user inputs invalid file names when prompted for input files.

(3) **View:** Write code to read and print the contents of an input file to the console. This can be used for viewing YazLang output. (Even though you can't intepret YazLang programs yet, you can test this on any input file you like, including a YazLang program itself.)

(4) **Menu/Reprompting:** Add code to allow the user to select whether to intepret, view, or quit and to reprompt if invalid mode is chosen or if an input file does not exist. Since you haven't implemented the "interpret" functionality yet, we suggest simply including the file prompting and `"YazLang interpreted and and output to a file!"` println when the user selects "interpret" for now.

(5) **Interpret a single command:** Write code to execute a single YazLang command and print the results to the console. You may want to begin by hard-coding the command and changing it as you debug. You can then move on to reading a single command from an input file. We recommend approaching the commands one at a time and in the order the appear in the above table (`CONVERT`, then `RANGE`, then `REPEAT`).

(6) **Intepret a YazLang file:** Modify your code to read each line from a YazLang input file, execute the command, and print the output to the console. Hard-code the file name for now.

(7) **File output:** Modify your code to produce output to a file instead of the console. Again, hard-code the file name for now.

Hints

The following suggestions and hints may help you be more successful on this assessment:

- When reading input from a file, you may need to use a mixture of line-based and token-based processing as shown in class and described in chapter 6 of the textbook. (The `FindMinAndMax`, `RateMovies`, and `Payroll` programs from class will be particularly helpful.)
- To check if a file exists, you should use methods from the `File` class. The textbook describes an alternate technique for dealing with missing files using `try/catch` statements, but you should **NOT** use this approach on this assessment (`try/catch` statements are considered "forbidden features" in this class).
- You may find the `startsWith` method of the `String` class useful for determining which type of command you are processing.
- You may also find the `replace` method of the `String` class useful for replacing occurrences of one character with another. For example, the code:

  ```
  String str = "mississippi";
  str = str.replace("s", "*");
  ```

  will result in the string `str` containing the value `"mi**i**ippi"`.
- The output from `RANGE` should end with a space—you do not need to use a fencepost approach for the output of this command.
- If your program is generating `InputMismatchException` errors, you are likely reading the wrong type of values from your `Scanner` (for example, using `nextInt` to read a string).
- If your program is generating `NoSuchElementException` errors, you are likely attempting to read past the end of a file or line.

Debugging Tips

You may want to initially "hard-code" the input and output filenames; in other words, you may want to just use fixed file names in your code rather than prompting the user to enter the file names. You may also want to temporarily print extra "debug" text to the console while developing your program, such as printing each command or argument as you read it. Be sure to remove this extra output along with any hard-coded file names before submitting your program.

It is easier to debug this problem using a smaller input file with fewer commands and arguments. The file `simple.txt` on the course website has a short YazLang program that will be useful for testing your program at first.

Implementation Guidelines

User Input/File Input

All console input should be processed using a `Scanner` and **should be read using the `nextLine` method only**. All file input should be processed using a `File` object and a `Scanner` as shown in class. File output should be performed using a `File` and a `PrintStream` as shown in class.

Be sure to use `nextLine` for *all* console input.

When interpreting a YazLang program, your program should break the input into lines and then into tokens using Scanner objects so that you can identify the command and look for all its arguments. Follow the process demonstrated in class and in the textbook.

You may assume that each line of any input file provided to the `I` option will contain a valid YazLang command (see below). There will not be any blank lines or lines that are not YazLang commands in these input files. You do not need to check the file name or extension, and you should not assume that the file name or extension will have any particular format. In particular, **DO NOT** assume that all input files will end with `.txt`.

You may assume that whenever a YazLang command is expected, it will be valid. Specifically, you may assume that:

- each command will be on its own line
- the first word on each line will be a valid YazLang command (`CONVERT`, `RANGE`, or `REPEAT`)
- each command will have an appropriate number of arguments
- all arguments will be of the correct type and will meet the requirements outlined above

Permitted Java Features

For this assessment, you are restricted to Java concepts covered in chapters 1 through 6 of the textbook. In particular, you **ARE NOT** allowed to use arrays on this assessment. In addition, **you may not use the `repeat` method in the Java `String` class**. You must implement this functionality yourself for the YazLang `REPEAT` command.

Code Quality Guidelines

In addition to producing the desired behavior, your code should be well-written and meet all expectations described in the grading guidelines and the Code Quality Guide. For this assessment, pay particular attention to the following elements:

Capturing Structure

Your `main` method in this program may have more code than it has in previous assessments. In particular, you may include a limited amount of output and some control flow constructs (e.g. a loop to drive the menu) in `main`. However, your `main` method must remain a concise summary of your program's structure, and you must still utilize methods to both capture structure and eliminate redundancy.

Each method should perform a single, coherent task, and no method should do too much work. To receive full credit, your program must include a separate method to execute each type of command, plus four (4) other non-trivial method besides `main`. (Therefore, your program should have a total of at least seven (7) non-trivial methods.)

Your program *must* include a single method to process each type of YazLang command.

Using Parameters and Returns

Your program should be a well-structured program as described above, utilizing parameters and returns as necessary. Your methods should not accept unnecessary or redundant parameters. In particular, your program should include only a single `Scanner` connected to `System.in`, though you may have additional `Scanner`s as well. You can (and probably should) use objects (such as `Scanner`, `File`, or `PrintStream`) as parameters and/or return values.

Code Aesthetics

Your code should be properly indented, make good use of blank lines and other whitespace, and include no lines longer than 100 characters. Your class, methods, variables, and constant should all have meaningful and descriptive names and follow the standard Java naming conventions. (e.g. `ClassName`, `methodOrVariableName`, `CONSTANT_NAME`) See the Code Quality Guide for more information.

Commenting

Your code should include a header comment at the start of your program, following the same format described in previous assessments. Your code should also include a comment at the beginning of each method that describes that methods behavior. Method comments should also explicitly name and describe all parameters to that method and describe the method's return value (if it has one). Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implementation details (such as describing loops or expressions included in the code). See the Code Quality Guide for examples and more information.

Running and Submitting

You can run your YazIntepreter program by clicking the "Activate the Terminal" message in Ed. (Note that this is a different process than previous assessments.) You may also need to click the arrow at the bottom of the window to reveal the terminal. This will compile and execute your code and show you any errors, or the output of your program if it runs correctly. If you believe your output is correct, you can submit your work by clicking the "Mark" button in the Ed assessment. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA.**

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. Note the due date and time carefully—**work submitted after the due time will not be accepted**.

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from lessons, section, and lab
- Reading the textbook
- Visiting support hours
- Posting a question on the message board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.

Reflection

In addition to your code, you must submit answers to short reflection questions. These questions will help you think about what you learned, what you struggled with, and how you can improve next time. The questions are given a `YazInterpreter Reflection` slide in the Ed lesson; type your responses directly into those textboxes.