Comprehensive Practice

Section Two

Goal

- We are going to write a program that evaluates fully parenthesized arithmetic expressions using the standard operators for addition, subtraction, multiplication, and division, and a special operator for exponentiation ("^").

Expression Evaluator

- For example, if we wanted to evaluate this expression:

  $$18.4-2.3\times 8.519.5+2.74.9$$

- We would write it as a fully parenthesized expression as:

  $$(18.4–((2.3*8.5)/(19.5+(2.7\verb|^|4.9))))$$

- We have conventions that allow us to leave out many of the parentheses in this expression
  - but it is very challenging to process such expressions because then we have to handle precedence issues.
  - We will keep it simple by always requiring parentheses for every operator.

- Normally, we assumed that spaces would be used to separate all of the individual tokens.
  - That allowed us to use a Scanner for reading the tokens
  - but it can be very annoying to have to include spaces for each individual element.
  - For example, the expression above would have to be rewritten as:

    (18.4 - ((2.3 * 8.5) / (19.5 + (2.7 ^ 4.9))))
- So we will write a more sophisticated version this time that allows you to have as little or as much spacing as you'd like.

- We also should be careful to report most errors if the user leaves off parentheses or doesn't have them matched properly.
  - This error checking is a nice complement to the other operations we will be performing to evaluate these expressions.
- But we will avoid one thorny issue.
  - We won't allow numbers to have a minus sign in front of them.
  - So even though it makes sense to form an expression like this:

    (-2.5/4.5)

  - We won't allow that minus sign in front of the 2.5.
  - Instead you'd have to form an expression like:

    ((0-2.5)/4.5)

How can we do it?

- We will develop the program in two stages:
  - Splitting into tokens
    - We have seen that the Scanner class can be used to tokenize a string using whitespace
    - but we want to allow a user to leave out the whitespace and still have his or her input properly tokenized.
  - **The Evaluator**
    - Now that we are allowed to read the tokens of a string, we can work on the code that will evaluate the tokens that we find in a fully parenthesized expression.

Splitting into tokens(1)

- We are going to build a supporting class called StringSplitter that will solve this fairly specialized task.
  - It will behave somewhat like a Scanner, but it won't require whitespace to break apart the string.
  - We can use the standard method names from the Scanner class for getting a next token and asking if there is another token.
  - We should also introduce a peek method that allows the client to ask about the next token without actually reading it.
  - And we will need a constructor that takes the string to split as a parameter.

Splitting into tokens(2)

1. Array
2. ArrayList
3. LinkedList
4. Stack
5. Queue
6. Binary Tree
7. Hash Table
8. Priority Queue

- So the public methods will be:
  - public StringSplitter(String line)
  - public boolean hasNext()
  - public String next()
  - public String peek()
- This task is going to involve：
  - Scanning through the letters of a string from beginning to end
  - And returning to the client of the class the individual tokens that we find
- Our class has to keep track of：
  - What has been read already
  - And what it has left to read.
  - It will also require a lot of peeking ahead to figure out how to separate the individual characters into tokens.
- For example, if the string is "(2.3*4.8)“
  - We will want to produce this sequence of tokens:
    "(", "2.3", "*", "4.8", ")"
  - As we are reading a number like 2.3, we will need to peek ahead to see what comes next so that we can recognize the end of the number.
- What data structure shall we use for solving this problem?

Splitting into tokens(3)

- It turns out that a queue is a great structure for solving this problem.
  - It allows us to examine all of the characters from beginning to end
  - And it keeps track of what we have already looked at and what we have left to look at.
  - It also allows us to peek ahead.

Splitting into tokens(4)

- What's the big difference between our splitter and a simple Scanner?
  - We need to recognize certain special characters as being tokens.
    - If we encounter a parenthesis or one of the arithmetic operators, we have to make that a token whether or not it is surrounded by whitespace.
    - It is best to include these special characters in a string constant for the class.
- In terms of fields, since we are reading a character at a time
  - We will want a Queue<Character> to store the individual characters of the string
  - A field for storing the next token.
  - Because we want to allow the client to peek ahead, we should have the field for the token always store the next token to be processed
    - that way we can allow the client to peek at it as much as he or she wants without actually reading anything

Splitting into tokens(5)

- We have to provide the client with a way to ask whether there is a next token to be processed.
- We could use an extra field for this, but we could also just set the token field to null when there are no tokens left to process.
- Here is a basic outline of our class including our two fields
  - Our special constant
  - and a constructor that adds the letters of the string to our queue of characters:

Splitting into tokens(5)

```
public class StringSplitter {
        private Queue<Character> characters;
        private String token;
        public static final String SPECIAL_CHARACTERS = "()+−*/^";

        public StringSplitter(String line) {
                characters = new LinkedList<Character>();
                for (int i = 0; i < line.length(); i++) {
                        characters.add(line.charAt(i));
                }
                findNextToken();
        }
        ...
}
```

Splitting into tokens(6)

- Notice that the constructor ends with a call on a method called findNextToken.
  - The idea is that we want a private method that will process the queue and give an appropriate value to the field called token.
  - Most of the work of writing this class is to write that method.
- To write the findNextToken method, we have to consider all of the possible cases and make sure that we handle each one.
  - We want to skip whitespace just as the Scanner does, so we will have to include code to do that.
  - After skipping any leading whitespace, we would normally be ready to build up the next token.
  - But we have to consider the case where we run out of characters, in which case there are no more tokens to produce.
  - If we do have a token to produce, we'll have to remove characters one at a time from the queue and add them to the token that we are building up.

Splitting into tokens--findNextToken

- The basic approach can be described with the following pseudocode:

```
skip whitespace.
if (nothing left) {
  token = null;
} else {
        initialize token to the next queue character.
        while (next queue character is part of this token) {
                token = token + (next character from the queue).
        }
}
```

- To skip the leading whitespace, we can peek ahead in the queue to see if the next character is a whitespace character.
- A helpful static method from the Character wrapper class called isWhitespace that returns true if a given character is whitespace such as a space, tab, or line break.
- We can't peek ahead when the queue is empty, so we have to include a special test for that as well:

Splitting into tokens- building up a token(1)

```
while (!characters.isEmpty() &&
                Character.isWhitespace(characters.peek())) {
        characters.remove();
}
if (characters.isEmpty()) {
      token = null;
} else {
      ...
}
```

- Now we need to write the code for building up a token character by character.
  - We can initialize the token to the next character in the queue by saying:
    - token = "" + characters.remove();

Splitting into tokens- building up a token(2)

- Then we run into the problem of knowing how many more characters to include in this token.
  - We have a set of special characters that are supposed to be one-character tokens
    - so if the token is any of those, then we need to stop adding characters to the token.
  - We can use our string constant and a call on contains to handle that special case:

```
if (!SPECIAL_CHARACTERS.contains(token)) {
        ...
}
```

Splitting into tokens- building up a token(3)

- Now we need a loop that will append characters to the current token until it finds something that isn't part of the token.
  - We would want to stop if we came across a whitespace character.
  - But we would also want to stop if we came across any of the special characters.
  - And we have an extra problem in that we might run out of characters completely if this is the last token to be processed.
- The logic gets fairly complicated in this case, so it is helpful to introduce a `boolean` variable that keeps track of whether or not we are done:

Splitting into tokens- building up a token(4)

```
// a boolean variable that keeps track of whether or not we //are
done
boolean done = false;
while (!characters.isEmpty() && !done) {
    char ch = characters.peek();
    if (Character.isWhitespace(ch) ||
            SPECIAL_CHARACTERS.indexOf(ch) >= 0) {
                done = true;
    } else {
                token = token + characters.remove();
            }
}
```

- This completes the private method for finding the next token

The Evaluator

- We have a support class that will allow us to read the tokens of a string
- We can work on the code that will evaluate the tokens that we find in a fully parenthesized expression.
- We are going to implement a variation of a famous algorithm known as the shunting-yard algorithm that was invented by Edsger Dijkstra.
  - It uses two stacks to save intermediate results.
  - One stack stores numbers and the other stack stores symbols.

- The basic idea is that we store values in the two stacks until we are ready to process them.
- As we see left parentheses and operators, we push them onto the symbol stack.
- As we see numbers, we push them onto the number stack.
- And when we see a right parenthesis, we know we have all of the information for a given sub-expression and we go ahead and evaluate it.
- We then push the result back onto the number stack.

- Consider a simple case of evaluating "(2+3)".
- The following Table shows how the initially empty stacks have elements added to them until we encounter the right parenthesis
  - at which point we evaluate the sum and push the result onto the number stack.

Evaluation of "(2+3)"

Token	Action	Symbol Stack	Number Stack
		[]	[]
(	Push onto symbol stack	[(]	[]
2	Push onto number stack	[(]	[2.0]
+	Push onto symbol stack	[(,+]	[2.0]
3	Push onto number stack	[(,+]	[2.0,3.0]
)	Evaluate expression and push result onto number stack	[]	[5.0]

- Notice that the overall value is the one and only value stored in the number stack when we are done
  - and the symbol stack is empty when we are done.
  - If there are other values left in either stack, then we know that we had an illegal expression.
- We can form complex sub-expressions that need to be evaluated as well.
  - For example, what if the expression to evaluate had been "((4/2)+(7-4))"?
  - This expression will have the same value
  - With the two stack approach, we can keep track of each part of this expression until we are ready to process it.

Evaluation of "((4/2)+(7-4))"

Token	Action	Symbol Stack	Number Stack
		[]	[]
(	Push onto symbol stack	[(]	[]
(	Push onto symbol stack	[(, (]	[]
4	Push onto number stack	[(, (]	[4.0]
/	Push onto symbol stack	[(, (, /]	[4.0]
2	Push onto number stack	[(, (, /]	[4.0, 2.0]
)	Evaluate expression and push result onto number stack	[(]	[2.0]
+	Push onto symbol stack	[(, +]	[2.0]
(	Push onto symbol stack	[(, +, (]	[2.0]
7	Push onto number stack	[(, +, (]	[2.0, 7.0]
2	Push onto symbol stack	[(, +, (, 2]	[2.0, 7.0]
4	Push onto number stack	[(, +, (, 2]	[2.0, 7.0, 4.0]
)	Evaluate expression and push result onto number stack	[(, +]	[2.0, 3.0]
)	Evaluate expression and push result onto number stack	[]	[5.0]

- To write the code, we just have to implement the algorithm.
  - It is easiest if we assume the input has no errors
  - but it's better to recognize errors when we can
  - We can't really recover from an error
    - it can get complex to report the nature of each error.
  - So let's strike a middle ground
    - recognizing as many errors as we can
    - but giving just a simple error message if we encounter a mistake.

- We know that we want to process tokens until we either encounter an error or run out of tokens.
- It is helpful to introduce a boolean flag that keeps track of whether an error has been seen.
- We expect to reach a point where the symbol stack is empty and the number stack has exactly one value in it.
  - In that case, we could report that one number as the overall result.

- The basic structure of our solution is:

```
StringSplitter data = new StringSplitter(line);
Stack<String> symbols = new Stack<String>();
Stack<Double> values = new Stack<Double>();
boolean error = false;
while (!error && data.hasNext()) {
        ...
}
if (error || values.size() != 1 || !symbols.isEmpty()) {
                System.out.println("illegal expression");
} else {
        System.out.println(values.pop());
}
```