Overview

Reverse Polish Notation (or postfix notation) is a mathematical notation in which operators follow operands. For example, the infix expression 2 + 4 is expressed as 2 4 + in postfix notation, and 1 + 4 * 3 is expressed as 1 4 3 * +. In this assignment, you are required to develop a reverse polish notation calculator that can take an infix expression, convert it to postfix notation, and then evaluate the expression to solve the equation.

Assumed Knowledge

Infix expressions are made up of operands and operators. Operands are the input: in the expression 1 + 2, the operands are 1 and 2. Operators are the action: for example, adding, subtracting, multiplying or dividing. The order operators are evaluated matters: choosing to do an addition before a multiplication gives a different result than doing the multiplication before the addition.

As an example, consider the expression 1 + 2 * 3. Evaluating the addition operator before the multiplication yields a different result:

Addition first:

1 + 2 * 3 == (1 + 2) * 3

== 3 * 3

== 9

Multiplication first:

1 + 2 * 3 == 1 + (2 * 3)

== 1 + 6

== 7

To avoid this ambiguity, operators are given precedence — when given a choice, operands are done in a specific order:

- Brackets (Parentheses)
- Exponents
- Division / Multiplication
- Addition / Subtraction

Thus, according to our precedence rules, the correct answer to our example above is 7 (multiplication before addition).

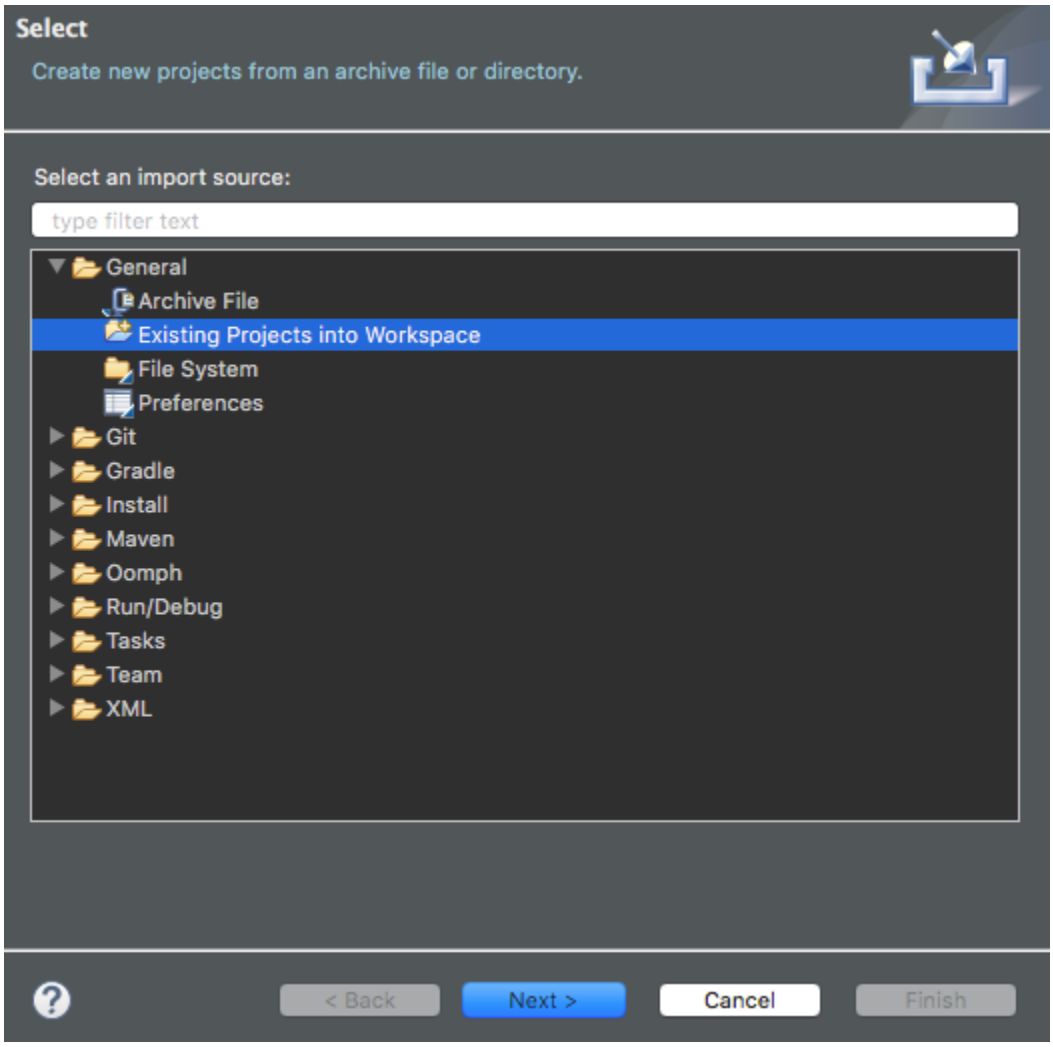
Implementation

There are two parts to this assignment. In part 1, you are required to implement several collections (Linked List, Stack and Queue). In part 2 you will implement an infix-to-postfix parser, and a postfix calculator. This assignment has tests to show how well your solution is working. **These tests are NOT 100% complete, meaning they will miss things, so you need to do your own testing as well. Final marking for the assignment will use additional tests, meaning if your code may pass these tests, but still fail the final marking if your code is not fully functional.**

Importing into eclipse

The Assignment has been provided as an eclipse project. You just need to import the project into an existing workspace. In the latest version of Eclipse, go File > Open Projects from File System and

open the directory of the project (the one with the src folder in it). For older versions, see Figure 1 for a visual guide. Make sure that your Java JDK has been set, as well as the two jar files that you need for junit to function. This can be found in Project > Properties > Java Build Path > Libraries. The jar files have been provided within the project; there is no need to download any other version and doing so may impact the testing environment. Cleaning the project may help some issues with your initial running of the project. This can be found in Project > Clean.



Part 1: Collections

In part 1, you will re-implement some of the Java Collections classes: LinkedList, Stack and Queue. There are three classes that require implementation: DSList.java, DSStack.java and DSQueue.java.

There is an «interface» provided for List, and «abstract classes» for both Queue and Stack. You will use the Linked List implementation as the basis for implementing Stack and Queue. For some methods there may be additional comments in the Javadoc.

DSList.java

Marks:

DSList()	0
DSList(Node)	2

DSList(DSList)	6
remove(int)	5
indexOf(Token)	3
get(int)	3
add(int, Token)	3
contains(Token)	3
remove(Token)	10
add(Token)	4
isEmpty()	3
size()	3
toString()	5

DSList will extend the List class defined in List.java. The implementation will be a double-linked list and must implement the abstract methods from List.java.

DSList should have one data member: public Node head. Others can be added if you require them.

Implement the following methods in the List class:

- `Constructor:` implement a blank constructor which takes no parameters.
- `Constructor:` implement a constructor accepting one Node (containing a Token object). The constructor should set head to the given Node.
- `Copy constructor:` implement a copy constructor accepting a DSList object. The copy constructor should perform a deep copy of the DSList passed to the constructor: the new DSList should not contain references to the Node objects in the second DSList. (The two DSLists should be independent: changing the contents of Node objects in one DSList should not affect the other).
- `public boolean add(Token obj):` The add method should append the specified object to the end of the List.
- `public boolean isEmpty()`
- `public int size()`
- `public String toString():` this should return a String created by concatenating each Nodes toString(). A single space: ‘ ’ should be inserted between each Nodes toString(). No trailing space should be inserted. For example, if the list contains 3 Node objects, an appropriate toString() return value could be `‘1 2 3’`, but not `‘123’` or `‘1 2 3 ’` [note the trailing whitespace]. For further details, refer to the unit tests supplied with the assignment.
- `public boolean equals(Object other):` two DSList objects are equal if they contain the same Tokens in the same order.
- `public int hashCode()`

Implement the abstract methods in List.java. The Javadoc annotations in List.java explain what each methods should do.

- `public boolean contains(Token object)`
- `public boolean remove(Token object)`
- `public Token remove(int index)`
- `public Token get(int index)`

DSQueue.java

Marks:

`DSQueue()`	0
`DSQueue(DSQueue)`	2
`offer(Token)`	2
`poll()`	2
`peek()`	2
`isEmpty()`	0
`toString()`	0

The Queue implementation will extend the abstract class Queue.java. The base storage of the Queue will be a List: you'll use the implementation you did in DSList.java.

Implement:

- Constructor that accepts no parameters. This constructor should initialize the internal storage of the Queue.
- Copy constructor that accepts a Queue. This constructor should do perform a deep copy of the second Queue.

Implement the abstract methods in Queue.java. The Javadoc annotations explain what the methods are required to do:

- `public boolean offer(Token object)`
- `public Token poll()`
- `public Token peek()`

DSStack.java

Marks:

`DSStack()`	0
`DSStack(DSStack)`	2
`peek()`	2
`pop()`	2
`push()`	2
`empty()`	2
`toString()`	0

The Stack implementation will extend the abstract class Stack.java. The base storage of the Stack will be a List: you'll use the implementation you did in DSList.java.

Implement two constructors:

- `Constructor` that accepts no parameters. This constructor should initialize the internal storage of the Stack.
- `Copy constructor` that accepts a Stack. This constructor should do perform a deep copy of the second Stack. Implement the abstract methods in Stack.java. The Javadoc annotations explain what the methods are required to do.

Implement the following methods:

- `public boolean empty()`

- `public Token peek()`
- `public Token push()`
- `public Token pop()`

Part 2: Postfix

The second part of this assignment requires you to use the Collections to create an infix-to-postfix converter and a postfix notation parser.

Marks:

`infixToPostfix()`	10
`evaluate(Queue)`	10

Converting from Infix to Postfix: Shunting-Yard Algorithm

We can use Dijkstra's 'Shunting-Yard algorithm' to parse infix expressions and output a reverse polish notation. The shunting-yard algorithm is as follows:

- While there are tokens to be read:
    1. Read a token.
    2. If the token is a number: add it to the output queue.
    3. If the token is an operator o1:
        - While there is a token o2 at the top of the stack, and either:
            - o1 is left-associative and has a precedence equal to that of o2
            - o1 has precedence less than o2

          pop o2 off the stack and push o2 onto the output queue
        - push o1 onto the stack.
    4. If the token is a left parenthesis: push it onto the stack.
    5. If the token is a right parenthesis:
        - Until the token at the top of the stack is a left parenthesis:
            - Pop the top of the stack and add it to the output queue.
        - Pop the top of the stack (the left parenthesis), but not onto the output queue. If there are no tokens left on the stack but we did not find a left parenthesis, the parentheses are unmatched (input error).
- When there are no more tokens to be read:
    1. while there are tokens on the stack
        - If the token is a parenthesis, we have an input error (unmatched parentheses)
        - Pop the stack onto the output queue.
    2. Return the output queue.

Parsing a Postfix Expression

An algorithm to evaluate a postfix expression is as follows:

- While there are more input tokens:
    - Read the next token.
    - If the token is an operand:
        - Push it onto the stack.

      else the token is an operator O1:
        - Pop the top element of the stack into a temporary variable E2.
        - Pop the top element of the stack into a temporary variable E1.

        - Evaluate the expression E1 O1 E2, and push the result onto the stack.
- When there are no more input tokens, there should be one item in the stack (if there is more than one item in the stack, the input contained an error). Return the result.

Style and Comment Marks

Whilst no additional marks are awarded for style and commenting, up to five (5) marks may be removed for submissions with inappropriate style, lack of comments or bad coding practice. For tests where your submission contains a hard-coded solution just to pass a test, rather than actually write a proper solution, you may lose those marks.