UNIVERSITY AT ALBANY
State University of New York

COLLEGE OF ENGINEERING AND APPLIED SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

ICSI311 Principles of Programming Languages

Assignment 01 Created by Qi Wang

Table of Contents

Part I: General Information

- All assignments are individual assignments unless it is notified otherwise.
- All assignments must be submitted via Blackboard. No late submissions or e-mail submissions or hard copies will be accepted.
- Unlimited submission attempts will be allowed on Blackboard. **Only the last attempt will be graded.**
- Work will be rejected with no credit if
    - The work is late.
    - The work is not submitted properly (Blurry, wrong files, crashed files, files that can't open, etc.).
    - The work is a copy or partial copy of others' work (such as work from another person or the Internet).
- Students must turn in their original work. Any cheating violation will be reported to the college. Students can help others by sharing ideas and should not allow others to copy their work.
- Documents to be submitted:
    - **UML class diagram(s)** – created with Violet UML or StarUML
    - **Java source file(s) with Javadoc style inline comments**
    - **Supporting files if any** (For example, files containing all testing data.)

  Note: Only the above-mentioned files are needed. Copy them into a folder, zip the folder, and submit the **zipped** file. We don't need other files from the project.
- Students are required to submit a design, all the error-free source files with Javadoc style inline comments and supporting files. Lack of any of the required items or programs with errors will result in a really low credit or no credit.
- **Grades and feedback**: TAs will grade. Feedback and grades for properly submitted work will be posted on Blackboard. For questions regarding the feedback or the grade, students should reach out to their TAs first. Students have limited time/days from when a grade is posted to dispute the grade. Check email daily for the grade review notifications sent from the TAs. **Any grade dispute request after the dispute period will not be considered.**

Part II: Grading Rubric

The following includes, but not limited to, a list of performance indicators used for grading.

Student Learning Outcomes:

SLO8 Develop a deep and comprehensive understanding of the object-oriented paradigm

		Levels of Performance			
		UNSATISFACTORY	**DEVELOPING**	**SATISFACTORY**	**EXEMPLARY**
Performance Indicators	**Performance Indicator #1:** UML design	None/Not correct at all	Visibility, name, and type/parameter type/return type present for node or expression tree class.	Visibility, name, and type/parameter type/return type present for all classes with minor issues, class relationships are correct.	Visibility, name, and type/parameter type/return type present for all classes without issues, class relationships are correct.
		0	5	7	10
	Performance Indicator #2: Comments	None/Excessive	Some tags are correct. "What" not "Why", few	Most tags are correct. Some "what" comments or missing some	All tags are correct. Anything not obvious has reasoning
		0		4	5
	Performance Indicator #3: Variable/Method naming	Single letters everywhere	Lots of abbreviations	Full words most of the time	Full words, descriptive
		0		4	5
	Performance Indicator #4: Node class	None	As a correct generic inner class instead	A generic class and all methods present with minor issues. Most of the requirements are met.	A generic class and all methods present with no issues. All requirements are met.
		0	5	7	10
	Performance Indicator #5: Expression Tree class	None		All methods present with minor issues. Most of the requirements are met.	All methods present without issues. All requirements are met.
		0		15	20
	Performance Indicator #6: The driver program	None		Both the node class and the expression tree class are mostly well tested.	Both the node class and the expression tree class are well tested.
		0		4	5
	Performance Indicator #7: Use a recursive approach	None		At least in one method and there are minor issues.	At least in one method and it is correct.
		0		4	5
Total Points Awarded		**Total points**: x /60 **Converted total points:** x /60 * 50			
Feedback to the student					

Part III: Examples

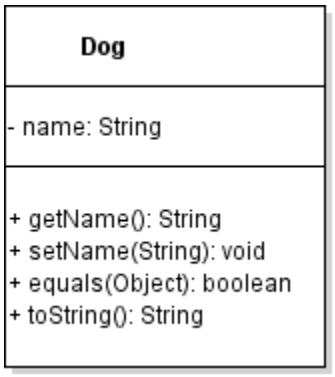
To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

Analysis-design-code-test/debug-documentation.

1) Read project description to understand all specifications(**Analysis**)
2) Create a design (an algorithm or a UML class diagram) (**Design**)
3) Create programs that are translations of the design (**Code/Implementation**)
4) Test and debug(**test/debug)**
5) Complete all required documentation. (**Documentation**)

The following shows a sample design and the conventions.

- *Constructors* and *constants* should not be included in a class diagram.
- For each *field* (instance variable), include *visibility*, *name*, and *type* in the design.
- For each *method*, include *visibility*, *name*, *parameter type(s)* and *return type* in the design.
    - DON'T include *parameter names*, only *parameter types* are needed.
- Show class relationships such as *dependency*, *inheritance*, *aggregation*, etc. in the design. Don't include the *driver* program or any other testing classes since they are for testing purpose only.
    - Aggregation: For example, if Class A has an instance variable of type Class B, then, A is an aggregate of B.



The corresponding source codes with inline Javadoc comments are included on next page.

```
import java.util.Random;

/**
 * Representing a dog with a name.
 * @author Qi Wang
 * @version 1.0
 */
public class Dog{
    /**
     * The name of this dog
     */
    private String name;

    /**
     * Constructs a newly created Dog object that represents a dog with an empty name.
     */
    public Dog(){
        this("");
    }

    /**
     * Constructs a newly created Dog object wit
     * @param name The name of this dog
     */
    public Dog(String name){
        this.name = name;
    }

    /**
     * Returns the name of this dog.
     * @return The name of this dog
     */
    public String getName(){
        return this.name;
    }

    /**
     * Changes the name of this dog.
     * @param name The name of this dog
     */
    public void setName(String name){
        this.name = name;
    }

    /**
     * Returns a string representation of this dog. The returned string contains the type of
     * this dog and the name of this dog.
     * @return A string representation of this dog
     */
    public String toString(){
        return this.getClass().getSimpleName() + ": " + this.name;
    }

    /**
     * Indicates if this dog is "equal to" some other object. If the other object is a dog,
     * this dog is equal to the other dog if they have the same names. If the other object is
     * not a dog, this dog is not equal to the other object.
     * @param obj A reference to some other object
     * @return A boolean value specifying if this dog is equal to some other object
     */
    public boolean equals(Object obj){
        //The specific object isn't a dog.
        if(!(obj instanceof Dog)){
            return false;
        }
        //The specific object is a dog.
        Dog other = (Dog)obj;
        return this.name.equalsIgnoreCase(other.name);
    }
}
```

Class comments must be written in Javadoc format before the class header. A **description** of the class, author information, and version information are required.

Comments for fields are required.

open {

TAB

TAB

Method comments must be written in Javadoc format before the method header. The first word must be a verb in title case and in the **third** person. Use punctuation marks properly.

open {

TAB

TAB

A description of the method, comments on parameters if any, and comments on the return type if any are required.

A Javadoc comment for a formal parameter consists of three parts:

- parameter tag,
- a name of the formal parameter in the design , (The name must be consistent in the comments and the header.)
- and a phrase explaining what this parameter specifies.

A Javadoc comment for return type consists of two parts:

- return tag,
- and a phrase explaining what this returned value specifies

More inline comments can be included in single line or block comments format in a method.

Part IV: Description

An algebraic expression tree
Goals:

- Review and develop a deep and comprehensive understanding of the object-oriented paradigm
- Review data structures such as lists, stacks, trees, etc.
- Review programming techniques such as recursion

A binary tree is a tree in which no node can have more than two children. An algebraic expression can be represented in a binary tree, an expression tree. The leaves of an expression tree are operands, such as constants or variable names, and the other nodes contain operators. This particular tree happens to be binary, because all the operators are binary, although this is the simplest case. It is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the unary minus operator.

In this assignment, you will design and implement an expression binary tree assuming that there are only four operations: addition, subtraction, multiplication, and division, all operands are positive numbers, and only Fully parenthesized algebraic expressions are used for testing.

To complete this assignment, you need to design a *node*, *an expression tree*, implement the design in Java, and test them in a *driver* program. Java is the required programming language for this assignment. If you need to review how to create an UML diagram or forget how to write comments in Javadoc style, see the examples in part III.

The node class:

A *node* contains a *generic* element, a reference to its left child and a reference to its right child. It can't be an *inner class* and it must have the following functionalities:

- overloading constructors
- *getters*/*setters*
- other methods if any
- overridden *equals*
- overridden *toString*

This node class must be a generic class. It will be used in the expression tree class or any classes where a node is required. When it is used in an expression tree, the element of a node can be an operand or an operator.

The expression tree class:

An expression tree object is a user-defined binary tree. It contains a reference to the tree (the *root*) and a reference to the associated expression- the *infix*. When requested, an expression tree can be visited in one of the three types of traversals, such as *preorder*, *inorder*, or *postorder*. The best design for the tree traversals is to design a tree iterator and use an iterator for traversals in an expression tree. To make it simpler for this assignment, we can simply include the traversal functionalities in the expression tree design. The following functionalities are required in this class:

- overloading *constructors*
- *getters*
- a preorder traversal method
- an inorder traversal method
- a postorder traversal method
- other methods if any
- overridden *equals*

- overridden *toString*

Constructors:

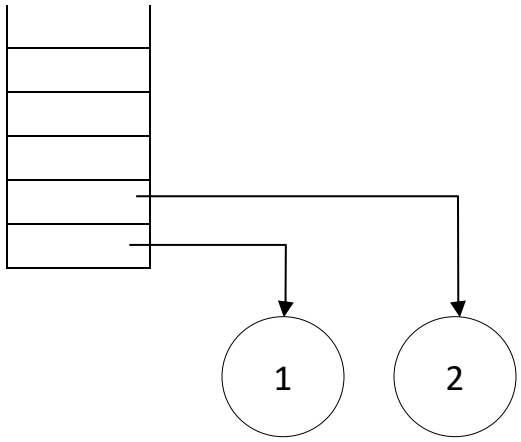
let's think about how to construct an expression tree. In another word, what are the designs for the constructors? Each class must have a *default* constructor in Java. The default constructor of this class should be provided although we will not use it in this assignment. A non-default constructor should be designed to accept a fully parenthesized infix expression and create a binary tree representation of it. It is hard to parse an infix expression since we must deal with balanced parentheses and consider the order of operations. One strategy is to convert the infix expression to a different form in which all the parentheses are removed, all operands are in the same order, and all operators are ordered by precedence. A postfix would be an appropriate form to use to construct an expression tree.

To construct a tree representation for a given an infix expression, we convert it to a postfix first. And then, we convert the postfix into an expression tree. We start with constructing left and right subtrees and attach them to their root. The method we describe next strongly resembles the postfix evaluation algorithm we learned. We scan the postfix expression by reading one token at a time. If the token is an operand, we create a one-node tree and push a reference to it onto a stack. If the token is an operator, we pop (references) to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right subtrees point to T_2 and T_1, respectively. A reference to this new tree is then pushed onto the stack. In summary, the non-default constructor will need to convert the infix expression to postfix, and convert the postfix to an expression tree, a tree representation of the infix expression.

The following example shows how to convert a postfix to a binary tree. Assume the following fully parenthesized infix expression is passed into the constructor. (Notice that the spacing is used to enhance readability. You can choose to have either no spacing or one space in between two adjacent tokens for this assignment.)

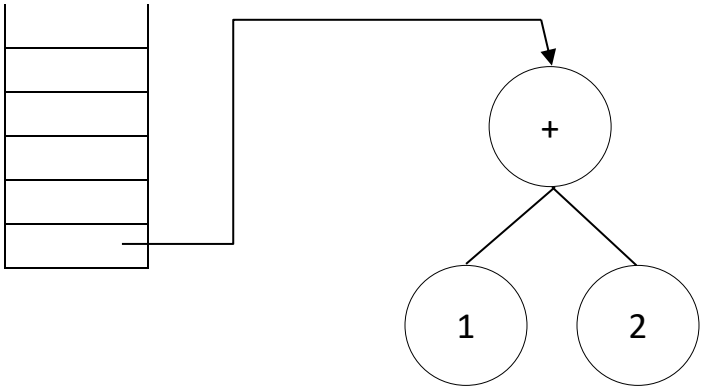
```
(1 + 2) * (3 * (44 + 55))
```

Its postfix is `1 2 + 3 44 55 + * *`. The first two tokens `1` and `2` are operands, so we create an one-node tree for each and push the references to them onto a stack.



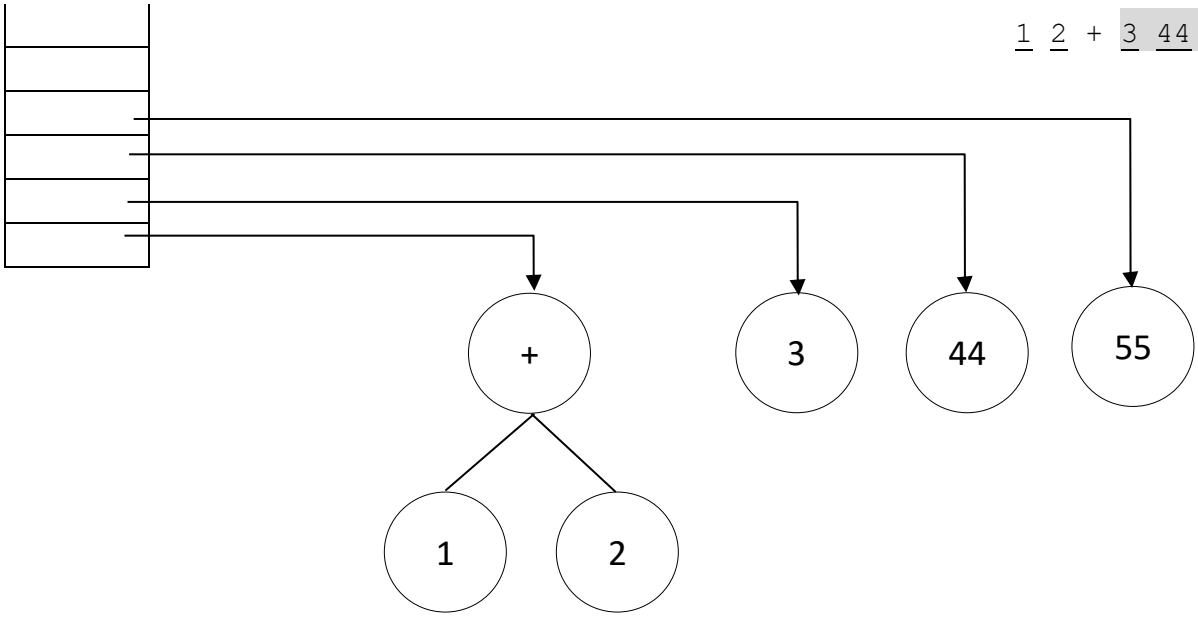
`1 2 + 3 44 55 + * *`

Next, `+` is read, two top tree references are popped, a new tree rooted at `+` is formed, and a reference to it is pushed onto the stack.



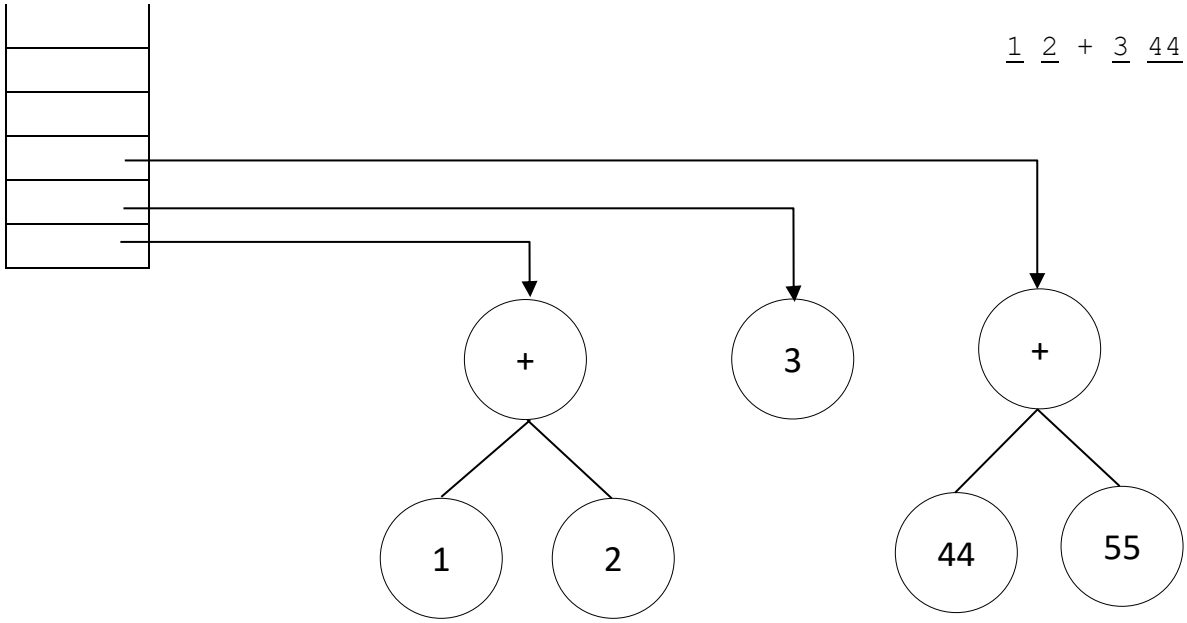
1 2 + 3 44 55 + * *

Next, 3, 44, and 55 are read, and for each an one-node tree is created and a reference to the corresponding tree is pushed onto the stack.



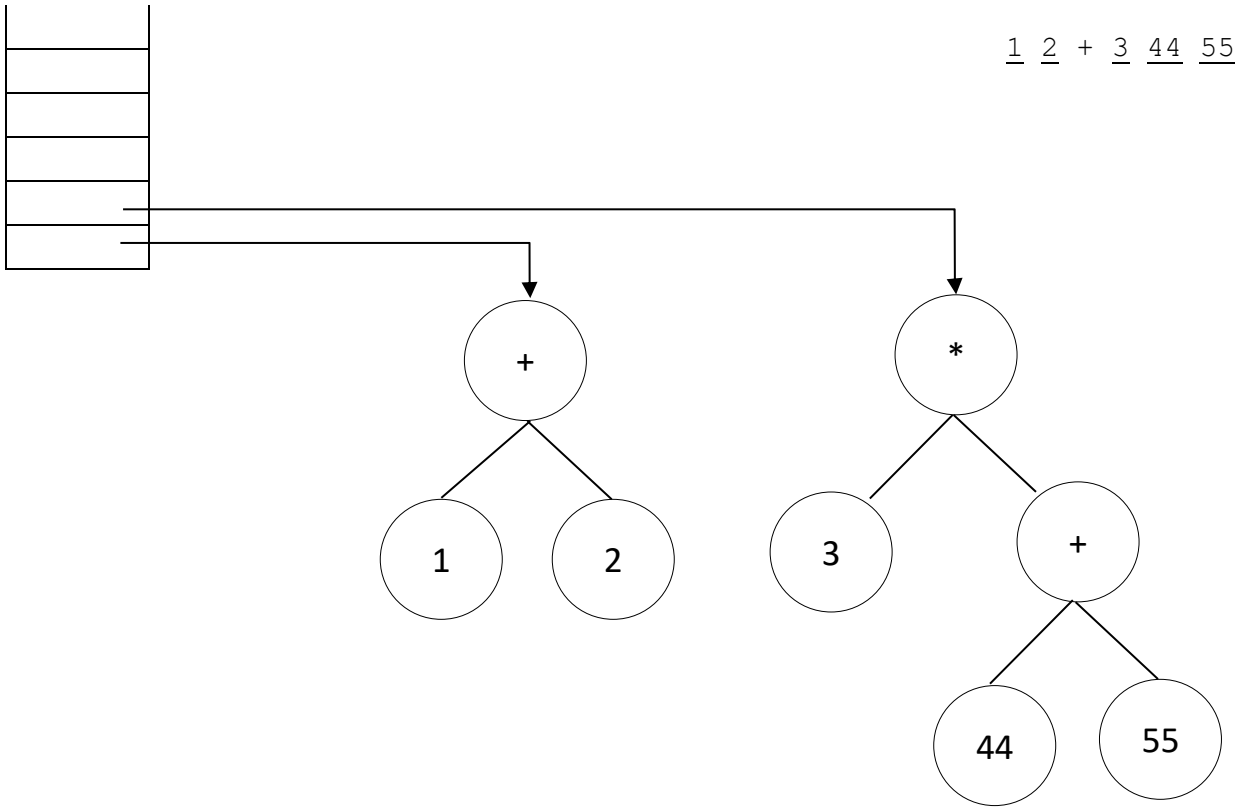
1 2 + 3 44 55 + * *

Next, + is read, two top trees are popped and merged. The reference to the new tree is pushed onto the stack.

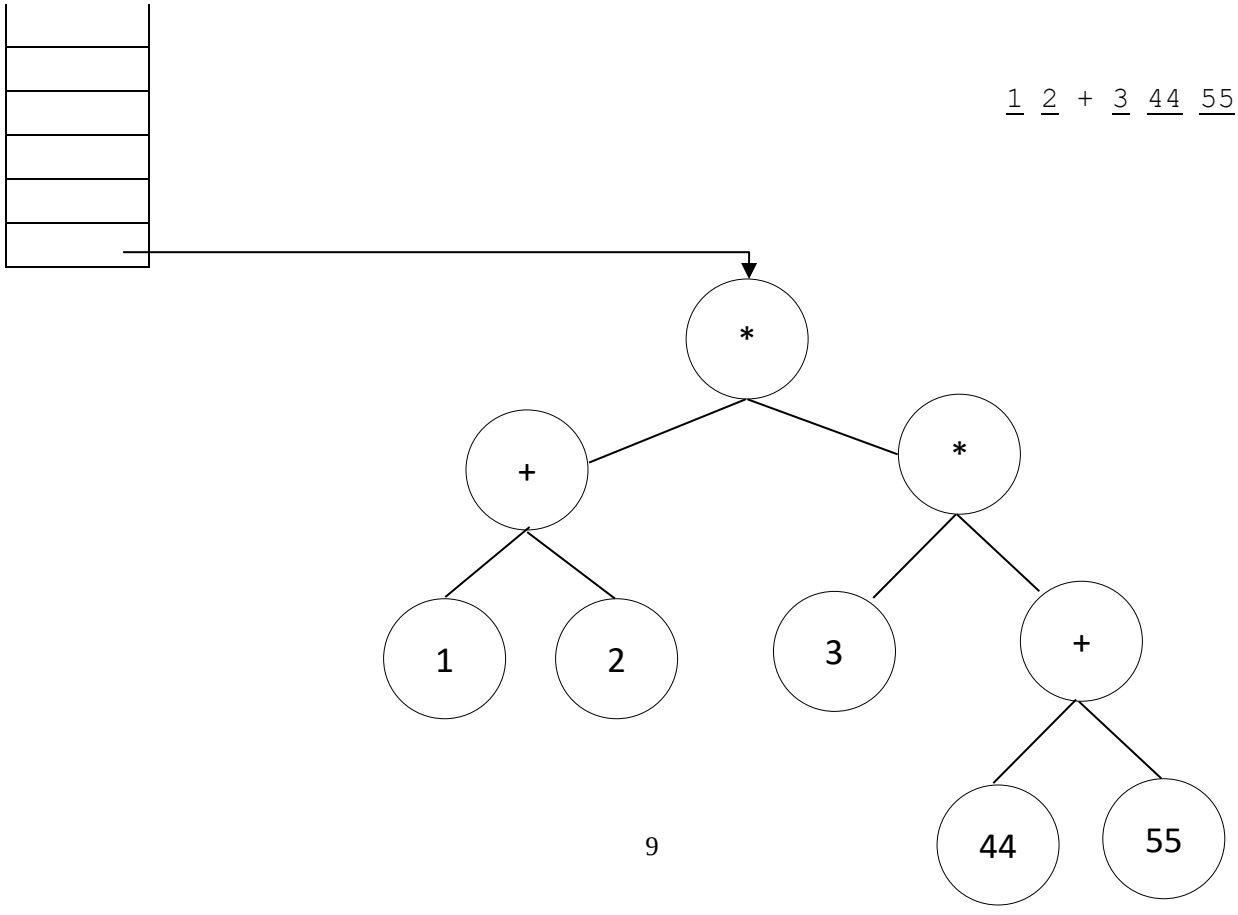


1 2 + 3 44 55 + * *

Next, * is read, two top trees are popped and merged. The reference to the new tree is pushed onto the stack.



Finally, the last token * is read, two trees are popped and merged, and a reference to the final tree is pushed onto the stack. This last token is the root of the tree.



Getters/Setters:

Getters should be provided for accessing the infix or the tree.

The expression tree class is used to represent an expression in a binary tree. We will not modify either the expression or its tree representation, therefore, no *setters* for this class.

Traversal methods:

Again, the best is to design a tree iterator, and use an iterator for traversals in an expression tree. To make it simpler for this assignment, we can simply include the traversal functionalities in the expression tree class.

These methods should use the *root* reference of this tree, visit each node in a particular order, store the nodes in the same order, and return a reference to the list of nodes. These methods can be designed using recursion.

Overridden *equals*:

An expression tree is equivalent to some other object if the other object is also an expression tree and both trees have the exact same nodes/elements in each location in the trees. This method can be designed using recursion.

Overridden *toString*:

A string representation of this expression tree should be returned from this method. For this assignment, the following information about this tree should be included in the returned string:

- the type of this tree: can be obtained by calling *getClass().getSimpleName()*
- the infix expression of this tree: already defined as an instance variable
- the prefix form of this tree: can be obtained from the corresponding traversal method
- the postfix form of this tree: can be obtained from the corresponding traversal method

This method can be designed using recursion.

Although recursive approach is not efficient, it is easy to write. It is also a good practice for this course. You are required to use recursion for some if not all the methods for which recursion is mentioned.

Other methods:

You may need to write some helper methods for some methods, such as the constructor and the traversal methods, in this class.

The driver program:

For testing, you must create at least 20 different fully parenthesized infix expressions in a text file, store the text file in the default folder of the project, and use this file for testing in *main*. The following is an algorithm for *main* method:

As long as there are more infix expressions in the file,

- Read the next infix expression and create a binary tree with it,
- Visit the tree in a preorder traversal
- Visit the tree in an inorder traversal
- Visit the tree in a postorder traversal
- Print the tree by invoking the toString method

When printing an expression tree using *print* or *println*, *toString* in the expression tree class is called implicitly. An expression tree should be printed by printing the returned string from *toString*. It is not required to print a visual tree that requires Graphical User Interface programming.

You need to write additional testing for *equals* method.
Here is the first infix expression you may use: `(1 + (2 * 3)) + (((4 * 55) + 66) * 77)`.

☺ Be Happy and Efficient!