UNIVERSITY AT ALBANY
State University of New York

COLLEGE OF ENGINEERING AND APPLIED SCIENCES
DEPARTMENT OF COMPUTER SCIENCE
ICSI213/IECE213 Data Structures

Project 03 Created by Qi Wang

Click here for the project discussion recording.

Table of Contents

Part I: General Project Information

- All projects are individual projects unless it is notified otherwise.
- All projects must be submitted via Blackboard. No late projects 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 project is late.
    - The project is not submitted properly (wrong files, not in required format, etc.). For example,
        - The submitted file can't be opened.
        - The submitted work is empty or wrong work.
        - Other issues.
    - The project 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, but not by allowing others to copy their work.
- Documents to be submitted as a zipped file:
    - **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.)
- Students are required to submit a design, all error-free source files with Javadoc style inline comments, and supporting files. Lack of any of the required items will result in a really low credit or no credit.
- Your TA will grade, and post the feedback and the grade on Blackboard if you have submitted the work properly and on time. If you have any questions regarding the feedback or the grade, please reach out to the TA first. You may also contact the instructor for this matter.

Part II: Project grading rubric

Components	**Max points**
UML Design (See an example in part II.)	Max. 10 points
Javadoc Inline comments (See an example in part II.)	Max. 10 points
The rest of the project	Max. 40 points

All projects will be evaluated based upon the following software development activities.

Analysis:

- Does the software meet the exact specification / customer requirements?
- Does the software solve the exact problem?

Design:

- Is the design efficient?

Code:

- Are there errors?
- Are code conventions followed?
- Does the software use the minimum computer resource (computer memory and processing time)?
- Is the software reusable?
- Are comments completely written in Javadoc format?
    a. Class comments must be included in Javadoc format before a class header.
    b. Method comments must be included in Javadoc format before a method header.
    c. More inline comments must be included in either single line format or block format inside each method body.
    d. All comments must be completed in correct format such as tags, indentation etc.

Debug/Testing:

- Are there bugs in the software?

Documentation:

- Complete all documentations that are required.

Part III: Examples on how to complete a project from start to finish

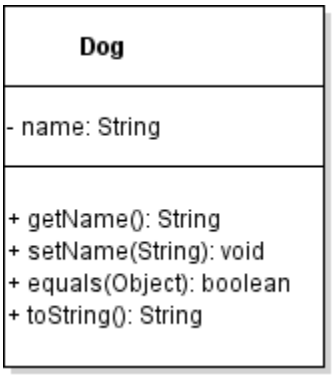
To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

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

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

The following shows a sample design. By convention.

- *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 and its helper class since they are for testing purpose only.



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.

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

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: Project description

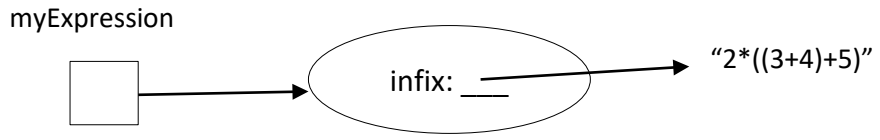
Project 3 An evaluator

In this project, you will create an arithmetic expression evaluator. Assume that all arithmetic expressions are in valid format. An arithmetic expression contains tokens such as arithmetic operators (addition, subtraction, multiplication, and division), integer operands, spaces and parentheses. For example, the first 5 expressions without spaces will result in 24. The last three expressions with spaces will result in 264, 264, 374, respectively.

2*((3+4)+5)
2*(3+(4+5))
2*((3+5)+4)
2*(3+(5+4))
2*((4+3)+5)
22 * ((3+4)+5)
22 * ((3 + 4) + 5)
22 * ((3 + 4) + 10)

To represent an arithmetic expression, *Expression* class needs to be designed. Each expression object contains an *infix*, a string. For example, the infix expression 2*((3+4)+5) can be represented as an *Expression* object like this

```
Expression myExpression = new Expression("2*((3+4)+5)");
```



Specifications:

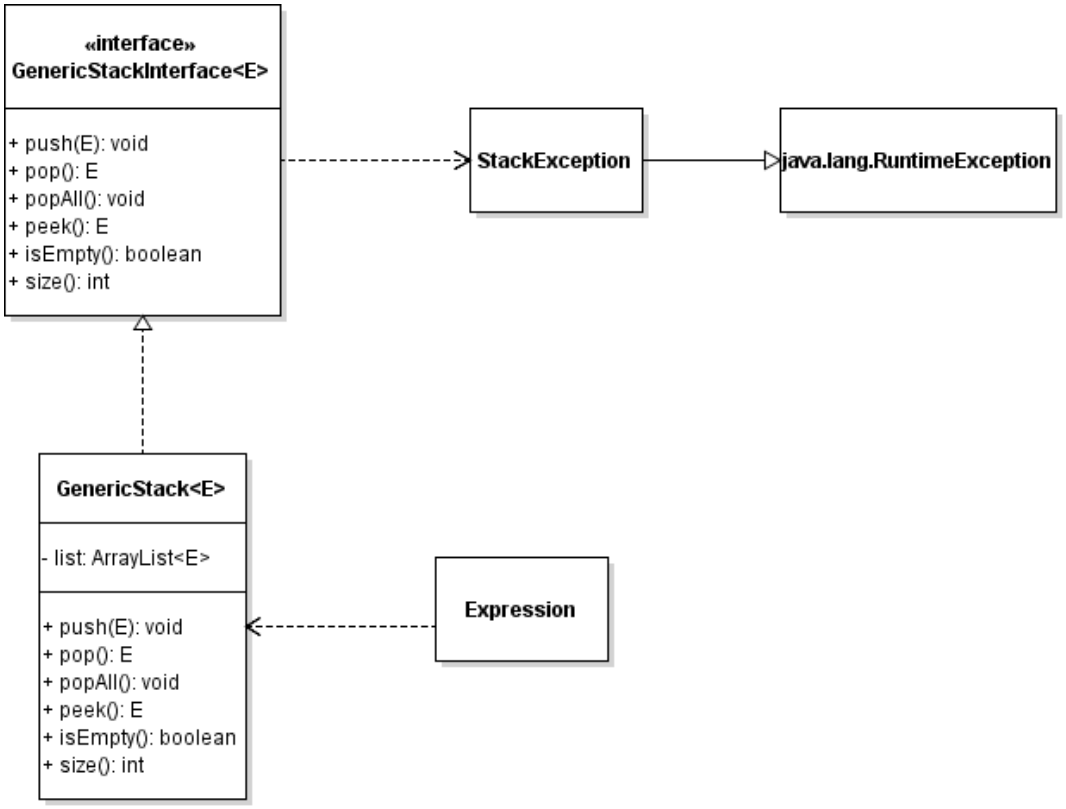
Here are what should be included in *Expression* class:

- An instance variable for the *infix*
- A default constructor
- A second constructor that takes a specific infix string
- Getter/setter for the *infix*
- An instance method that converts *this infix* to postfix and returns the postfix as a list of tokens
    - Use the *infix to postfix* algorithm we discussed in class. Use the *ADT stack* that you have completed from the previous lab.
    - When splitting an infix into tokens, you should not use *charAt* method and expect all operands are single-digit tokens. Instead you should split an infix by choosing proper delimiters. One way is to use operators, spaces and parentheses as delimiters and use them as tokens. Please check *StringTokenizer* class for proper methods that can be used.
- An instance method that evaluates *this infix* and returns the result. In this method,
    - First, you should convert this infix to postfix by calling the previous method.
    - And then, evaluate the postfix and return the result. Use the *postfix evaluation* algorithm discussed in class. Use the *ADT stack* that you have completed from the previous lab.

- Overridden *equals* and *toString*

Design:

Include your *Expression* design and *ADT stack* design completed for the lab in the same diagram.



Code/Test/Debug:

After implementing the Expression class, create an input file and add more expressions to the end of the list.

2*((3+4)+5)

2*(3+(4+5))

2*((3+5)+4)

2*(3+(5+4))

2*((4+3)+5)

22 * ((3+4)+5)

22 * ((3 + 4) + 5)

22 * ((3 + 4) + 10)

Write a simple driver to test your classes.

```
public static void main(String[] args) throws FileNotFoundException{

      //Create a scanner object to read the input file.

      //as long as there are more infix expressions,
```

```
//    -read one infix at a time, make an expression object.
//    -use the object reference to call the infix to postfix method and
//      evaluate method. And print the results.
```