



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

Table of Contents

Part I: General project information	02
Part II: Project grading rubric.....	03
Part III: Examples	04
Part IV: A. How to test?	06
B. Project description	07

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, files that can't open, etc.).
 - The project is a copy or partial copy of others' work (such as work from another person or the Blackboard site 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.)

Note: Only the above-mentioned files are needed. Copy them into a folder outside the Eclipse, 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 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 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.**
- **Proper use of the course materials including the source codes:** All course materials including source codes/diagrams, lecture notes, etc., are references for your study only. Any misuse of the materials is prohibited. For example,
 - *Copy the source codes/diagrams and modify them into the projects.* Students are required to submit the **original work** for the projects. **For each project, every single statement for each source file and every single class diagram for each design must be created by the students from scratch.**
 - *Post the source codes and diagrams on some Web sites.*
 - *Others*

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 Javadoc comments must be included before a class header.
 - b. Method Javadoc comments must be included before a method header.
 - c. More inline comments (in either single line format or block format) must be included 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 documents that are required.

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

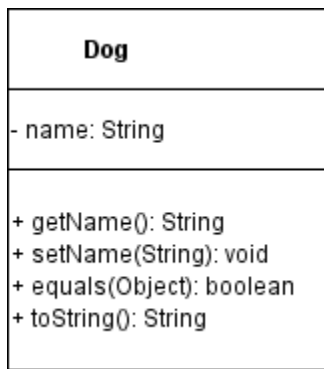
To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

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 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 with
     * @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 verb in title case and 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:

A. How to test?

There can be many classes in a software design. **A UML class diagram should contain the designs of all classes and the class relationships (For example, is-a, dependency, or aggregation).**

- **First, test each class separately in its own driver.**
 - Create instances of the class (If a class is abstract, the members of the class will be tested in its subclasses.).
For example, the following creates Dog objects.
Create a default Dog object.

```
Dog firstDog = new Dog();
```


Create a Dog object with a specific name.

```
Dog secondDog = new Dog("Sky");
```
 - Use object references to invoke instance methods. If an instance method is a value-returning method, call this method where the returned value can be used. For example, method `getName` can be called to return a copy of `firstDog`'s name.

```
String firstDogName;  
...  
firstDogName = firstDog.getName();
```

You may print the value stored in `firstDogName` to verify.

- If a method is a void method, invoke the method that simply performs a task. Use other method to verify the method had performed the task properly. For example, `setName` is a void method and changes the name of this dog. After this statement, the `secondDog`'s name is changed to "Blue".

```
secondDog.setName("Blue");
```

`getName` can be used to verify that `setName` had performed the task.

- Repeat until all methods are tested.
- **Next, test the entire design by creating a driver program and a helper class for the driver program.**
 - Create a helper class. In the helper class, at minimum three static methods should be included.

```
public class Helper{  
    //method 1  
    public static void start(){  
        This void method is decomposed.  
        It creates an empty list.  
        It calls the create method with a reference to the list passed to the method.  
        And then, it calls the display method with a reference to the list passed to the method.  
    }  
  
    //method 2  
    public static returnTypeOrVoid create(a reference to a list) {  
        This method creates objects using data stored in a text file and store the objects into the list.  
    }  
  
    //method 3  
    public static returnTypeOrVoid display(a reference to a list) {  
        This method displays the list of objects.  
    }  
}
```

- Create a driver program. In *main* of the driver program, call method *start* to start the entire testing process.

```
public class Driver{
    public static void main(String[] args){
        Helper.start();
    }
}
```

Notice that the driver and its helper class are for testing purpose only. They should not be included in the design diagram. But you will need to submit their source codes.

B. Project description

An expression evaluator

For 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)

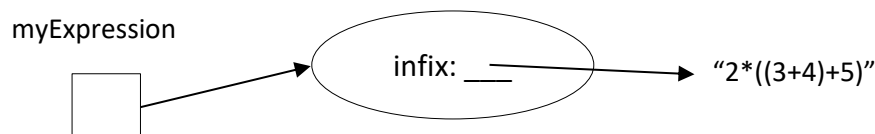
22 * ((3+4)+5)

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:

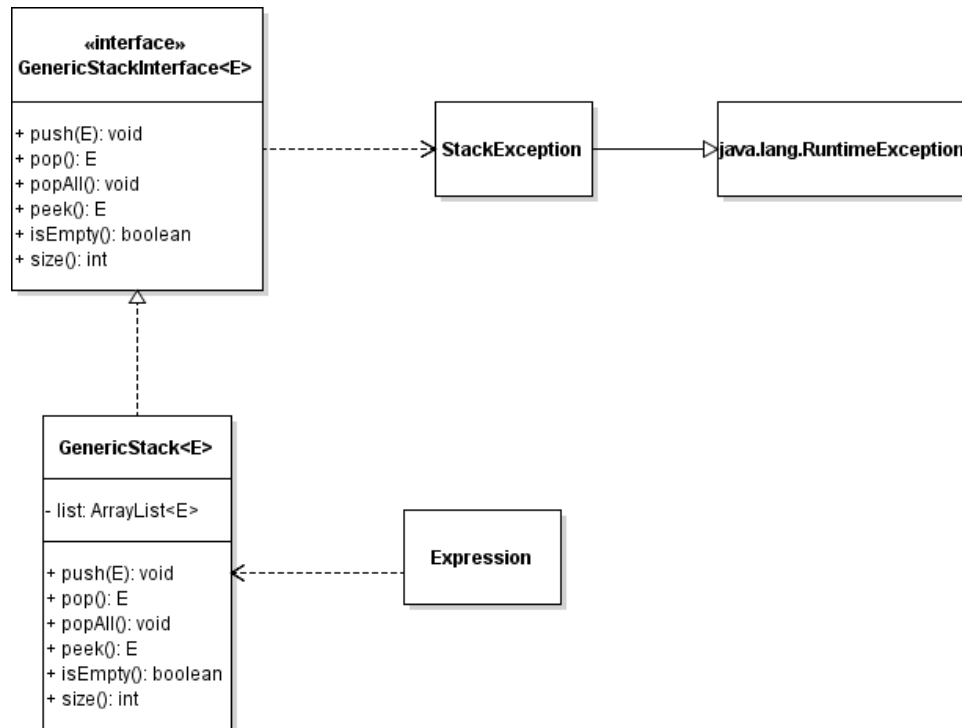
A generic ADT Stack:

To convert an infix to postfix or to evaluate a postfix, a stack is required. For this project, a generic *ADT Stack* implemented with an *array list* must be created and used. It is not allowed to use a Java JCF stack or any other stacks. The major operations of the generic ADT stack are shown on the next page.

An *Expression* class:

- An instance variable for the *infix*
- Overloading constructors
- Getter/setter for the *infix*
- The following two instance methods must use the *ADT stack* from the previous step.
 - An instance method that converts *this infix* to postfix and returns the postfix as a list of tokens

- Use the *infix to postfix* algorithm discussed in class.
- When splitting an infix into tokens, you should not use *charAt* method and expect all operands are single-digit tokens. Instead, you should split the 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.
 - First, call the previous method to convert this infix to postfix.
 - And then, evaluate the postfix and return the result.
 - Use the *postfix evaluation* algorithm discussed in class.
- Overridden *equals* and *toString*
- ...



Design:

Complete the design.

Code:

Implement all required classes properly.

Debug/Testing:

Note: It is required to store all testing data in a text file. It is required to use decomposition design technique.

To test the design, all operations must be tested. In general, a list of *Expression* object is created, and then, use the list to test other operations. It is not efficient to write everything in *main*. Method *main* should be small and the only method in a driver program. A helper class should be made to assist the driver. In the helper class, begin with three static methods. Method *start* creates an array list that will be used to store a list of expressions. And then, it calls *create* to add a list of *expression* objects to the array list. After that, it calls *displayAndMore* to print the list of expressions and more. More statements can be added to these three methods or more methods can be added to the helper class for testing. Method *main* should call *start* from the driver program to start the entire testing process.

- Create a helper class. In the helper class, at minimum three static methods should be included.

```
Public class Helper{
    //method 1
    public static void start(){
        This void method is decomposed.
        It creates an empty array list that can be used to store a list of expression objects.
        It calls create with a reference to the array list.
        - create adds a list of expression objects into the array list.
        It calls displayAndMore with a reference to the array list.
        - displayAndMore uses the list to test other methods.
    }

    //method 2
    public static returnTypeOrVoid create(A reference to an array list){
        Use the data stored in the text file to make expression objects.
        Add the expression objects into the array list.
    }

    //method 3
    public static returnTypeOrVoid displayAndMore(A reference to an array list){
        Print the list of expressions as both infix and postfix.
        Print the values of the list of expressions.
        ...
    }
}
```

- Create a driver program. In *main* of the driver program, call method *start* to start the entire testing process.

```
public class Driver{
    public static void main(String[] args){
        Helper.start();
    }
}
```

The sample testing file may contain items like this. 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)
```

Documentation:

Complete all other documents needed.