UNIVERSITY AT ALBANY
State University of New York

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

Project 01 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.
- Two 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.)

  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 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.
- **Grades and feedback**: TAs will grade. Feedback and grades for properly submitted work will be posted on Blackboard. 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. If students have any questions regarding the feedback or the grade, they should reach out to their TAs first.
- **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 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:

1. Are there bugs in the software?

Documentation:

2. Complete all documents that are required.

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

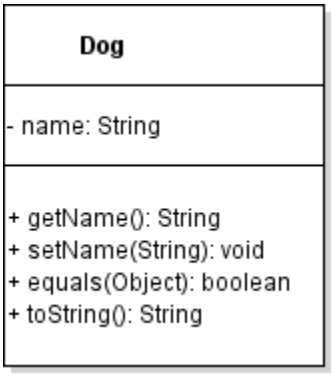
To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

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. 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 it is 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);
        }
}
```

TAB

TAB

open {

TAB

TAB

open {

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 a software design?

There can be many classes in a software design.

1. **First, create a UML class diagram containing the designs of all classes and the class relationships (For example, is-a, dependency or aggregation).**
2. **Next, test each class separately.**
   Convert each class in the diagram into a Java program. When implementing each class, a driver is needed to test each method included in the class design. In the driver program,
   i. Use the constructors to 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.
   ```
   Die secondDog = new Dog("Sky");
   ```

   ii. Use object references to invoke the 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.

   iii. 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.

   iv. Repeat until all methods are tested.

3. **And then, test the entire design by creating a driver program and a helper class of the driver program.**
   i. Create a helper class. In the helper class, 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 to add a list of objects to the list.
   And then, it calls the *display* method to display the list of objects.
   ```
   }
   ```
   //method 2
   ```
   public static returnTypeOrVoid create(parameters if any) {
   ```
   This method creates a list of objects using data stored in text files.
   ```
   }
   ```
   //method 3
   ```
   public static returnTypeOrVoid display(parameters if any) {
   ```
   This method displays a list of objects.

```
    }
}
```

ii. 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.

B. Project description

Project 1 Abstract Data Type(ADT) Bag

An ADT Bag is composed of a list of grocery items and a set of operations on the list. You may think of a grocery bag as an instance of the ADT Bag. A grocery bag contains a list of groceries. There is a set of operations that operate on the list. For example, we can add an item, remove an item, count the number of items, check for a specific item, etc. In this project, you will design an ADT bag by following software development cycle including specification, design, implementation, test/debug, and documentation.

Specification/Analysis:

The ADT Bag must contain the following operations:

- create an empty bag that can hold up to 100 items
- add an item to the end of the list of this bag- *insert(item)*
- remove the item at the end of this bag - *removeLast()*
- remove an item at a random index from this bag - *removeRandom()*
- get the index of the first occurrence of an item from this bag - *get(a reference to an item)*
- get a reference to an item at position index of this bag(get(index)),
- check how many items are there in this bag - *size()*
- check to see if this bag is empty - *isEmpty()*
- empty this bag - *makeEmpty()*

Design:

Complete a UML diagram to include all classes that are needed to meet the specifications. An interface class needs to be defined to specify the design of each operation. In the interface, what each operation does is specified. A class implementing this interface needs to be defined to specify how each operation does and a data structure is selected to store a collection of data.

Exceptions should to be considered when operations are designed. Java has two types of exceptions: checked exceptions and runtime exceptions.

Checked exceptions are instances of classes that are sub classes of *java.lang.Exception* class. They must be handled locally or explicitly thrown from the method. They are typically used when the method encounters a serious problem. In some cases, the error may be considered serious enough that the program should be terminated.

Runtime exceptions occur when the error is not considered as serious. These types of exceptions can often be prevented by fail-safe programming. For example, it is fairly easy to avoid allowing an array index to go out of range, a

situation that causes the runtime exception *ArrayIndexOutOfBoundsException* to be thrown. Runtime exceptions are instances of classes that are subclasses of the *java.lang.RuntimeException* class. *RuntimeException* is a subclass of *java.lang.Exception* that relaxes the requirement forcing the exception to be either handled or explicitly thrown by the method.

In general, some operations of an ADT list/ADT Bag can be provided with an index value. If the index value is out of range, an exception should be thrown. Therefore, a subclass of *IndexOutOfBoundException* needs to be defined.

Also, an exception is needed when the list/bag storing the items becomes full. A subclass of *java.lang.RuntimeException* should be defined for this erroneous situation. A full ADT bag should throw an exception when a new item is inserted. Similarly, an empty ADT Bag should throw an exception when removing an item from it.

Code/Implementation:

A class implementing the interface needs to be written to specify how each operation does and a data structure is selected to store a collection of data. An *array* must be used to store all items in an ADT Bag. The element type of the array should be *Object* type – the ultimate superclass of all other classes. When organizing items in the array, all items must be stored consecutively. This means shifting may be needed after some insertion or removal operations.

Javadoc comments need be written during this activity. Class comments must be included right above the corresponding class header. Method comments must be included right above the corresponding method header. Comments are also needed in each method to document each block of statements.

Debug/Testing:

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

To test the ADT bag design, all operations must be tested. In general, an empty ADT Bag is created, and then, fill the bag, display the list and 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 created to assist the driver. In the *Helper* class, minimum three static methods should be included. Method *start* creates an empty bag and is decomposed/calls other methods such as *create* and *display*. Method *create* should add items into the bag. Method *display* should display the items in the bag. More methods can be added to test other operations. Method *main* should call *start* from the driver program to start the entire testing process.

Test the entire design by creating a driver program and a helper class of the driver program.

- Create a helper class. In the helper class, 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 bag.
It calls *create* with a reference to the empty bag.
create adds a list of items into the bag.
It calls *display* with a reference to the bag.
display displays the list of items in the bag.

```
}
```

//method 2

```
public static returnTypeOrVoid create(A reference to a bag){
```

Using data stored in text files to make items.
Add the items into the bag.
Note: In this case, you may add *String* objects to an ADT Bag that can store items of *Object* type.

```
}
```

```
    //method 3
    public static returnTypeOrVoid display(A reference to a bag){
          Displays the list of items in the bag.
    }
}
```

- 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:

```
Apple
Pear
Orange
Milk
Bread
…
```

The path for this file should not start from a specific folder on your computer. It should start from the default folder. When grading the project, the testing file is expected in the default folder of the Java project in Eclipse.

Documentation:

Complete all other documents needed.