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

Project 01 by Qi Wang

The following parts are included in this document:

- Part I: General project information

- Part II: Project grading rubric

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

- Part IV: A. How to test?

    B. Project description

Proper use of the course materials including the source codes:

All course materials including source codes/diagrams, lecture notes, etc., are for your reference 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 (if any) must be created by the students from scratch.**
- Post the source codes, the diagrams, and other materials online.
- Others

Part I: General Project Information

- All work is individual work unless it is notified otherwise.
- Work will be rejected with no credit if
  - The work is not submitted via Duifene.
  - The work is late.
  - The work is partially or entirely written in Chinese.
  - The work is not submitted properly.
    - Blurry, wrong files, not in required format, crashed files, 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, but not by allowing others to copy their work.
- Documents to be included/submitted as a zipped folder:
  - 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.)

  Lack of any of the required items or programs with errors will result in a low credit or no credit.

- **How to prepare a zipped folder for work submission?**
  - Copy the above-mentioned files into a folder, rename the folder using the following convention:
    [*Your_first_name*][*your_last_name*]*ProjectNumber*
    - For example*, JohnSmithProject01*
  - Zip the folder. A zipped file will be created.
    - For example*, a file with name *JohnSmithProject01.zip* will be created*.
  - Submit the zip file on Duifene.
  - You must submit a project in this format. **Submissions not in the required format may be rejected or will result in a low credit or no credit.**

- **Grades and feedback**: Co-instructors will grade. Feedback and grades for properly submitted work will be posted on Duifene. For questions regarding the feedback or the grade, students should reach out to their co-instructors 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 co-instructors. **Any grade dispute request after the dispute period will not be considered.**

Part II: Project grading rubric

Project 1: (100 points) Performance Indicator (PI)	LEVELS OF PERFORMANCE INDICATORS				Points Earned
	UNSATISFACTORY	DEVELOPING	SATISFACTORY	EXEMPLARY	
PI1: (10 points) UML design	None/Not correct at all. (0)	Visibility, name, and type/parameter type/return type present only for some members for some classes. Some class relationships are incorrect. (<=5)	Visibility, name, and type/parameter type/return type present for each member for all classes with minor issues, class relationships are correct with minor issues. (<=7)	Visibility, name, and type/parameter type/return type present for each member for all classes without issues, class relationships are correct. (<=10)	
	Instructor's Comments:				
PI2(5 points): Comments (Javadoc format) All tags must be included correctly.	None (0)	Some comments are written properly. (<=2)	Most comments are written properly. (<=4)	All comments are written properly. (<=5)	
	Instructor's Comments:				
PI3(5 points): Variable/method naming	Single-letter names everywhere	Many abbreviations (<=2)	Full words most of the time (<=4)	Full words, descriptive (<=5)	
	Instructor's Comments:				
PI4(20 points): The interface	No methods are designed correctly. (0)	Some of the methods are designed correctly. Have a lot of issues. (<=5)	Most of the required methods are designed correctly. Have minor issues. (<=15)	All required methods are designed correctly. Have no issues. (<=20)	
	Instructor's Comments:				
PI5(30 points): The array-based implementation of the interface	No methods are implemented correctly. Have a lot of issues. (0)	Some of methods are implemented correctly. Have a lot of issues. (<=10)	Most of the required methods are implemented correctly. Have minor issues. (<=20)	All required methods are implemented correctly. Have no issues. (<=30)	
	Instructor's Comments:				
PI6(20 points):	No methods are tested correctly.	Some of the methods are tested correctly.	Most of the required methods are tested	All required methods are tested	

The driver program and its testing helper class	(0)	Have a lot of issues. (<=5)	correctly. Have minor issues. (<=15)	correctly. Have no issues. (<=20)	
	Instructor's Comments:				
PI6(10 points): User-defined Exceptions: <mark>- bag index out of bounds exception</mark> <mark>- bag exception</mark>	No exceptions are included or included correctly. (0)	One exception is included. Have issues. (<=5)	Both exceptions are included. Have minor issues. (<=7)	Both exceptions are included. Have no issues. (<=10)	
	Instructor's Comments:				
				Not submitted as a zipped folder	-10
				Total out of 100	
				Total out of 60	

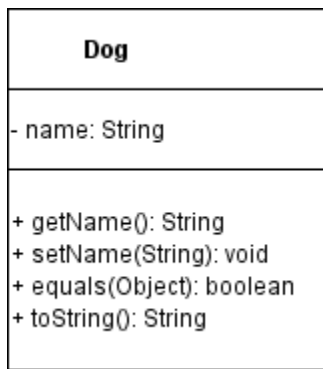**Part III: Examples on complete a project from start to finish**
To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

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.

```
            Dog

- name: String


+ getName(): String
+ setName(String): void
+ equals(Object): boolean
+ toString(): String
```

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

```java
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 verb in title case and in the **third** person. Use punctuation marks properly.

open {

open {

TAB

TAB

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.

6

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.
    - o   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");
        ```

    - o   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 firstDog's name. You may print the value stored in firstDogName to verify.

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

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

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

        Print the return value from getName to verify that the name has been changed correctly.

    - o Repeat until all methods are tested.

- Next, test the entire design by creating a driver program and a helper class for the driver program. When completing these two classes, you will test and learn how to use the classes (e.g., Dog class) that you have designed. You may choose different identifiers for the class names. For this project, Driver and Helper are used as the class identifiers.
    - o   Create a helper class for the driver. The helper class can begin with the following three static methods.
        ```
        public class Helper{
            public static void start(){
                    This void method is decomposed into a few tasks:
                    Create an empty list.
                    Fill the list by calling create with a reference to the list.
                    Print the list by calling display with a reference to the list.

                    …
            }

            public static returnTypeOrVoid create(a reference to a list) {
                    Create objects using data from an input file and store the objects into the list.

                     …
            }

            public static returnTypeOrVoid display(a reference to a list) {
                    Display the objects of the list.

                    …
            }
            …
        }
        ```

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

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

o There are more methods than what are invoked/tested by the previous three methods. You may add more statements in the methods or write additional methods to test the rest of the methods for all classes.

o **T**he 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

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. For this project, you will design an ADT bag that contains a list of items and operations on the list. An *Object* array, not an array list, must be used as the data structure. A project using a different data structure or a difference element type will result in 0 credit or low credit.

Specification/Analysis:

The ADT Bag must provide the following operations. The recommended identifiers are included.

- 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(an item to be searched)*
- get a reference to an item at position *index* of this bag - *get(index)*
- return the number of items in this bag - *size()*
- check if this bag is empty - *isEmpty()*
- empty this bag - *makeEmpty()*

Design:

Complete a UML diagram to include all classes excepting the testing classes (driver and helper). An interface class needs to specify the design of the operations. The interface specifies what each operation does. A class implementing this interface specifies how each operation is done. This class also selects a data structure to store a collection of data. Exceptions must be considered for some operations.

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 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

8

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.

Some operations in the ADT Bag require a parameter- an index value. If the index value is out of bounds, an exception should be thrown. Therefore, a subclass of *IndexOutOfBoundException* needs to be defined.

Also, an exception may occur when manipulating a full list/bag or an empty list/bag. For a full ADT bag, an exception should be thrown when inserting a new item into the bag. Similarly, for an empty ADT Bag, an exception should be thrown when removing an item from the bag. A subclass of *java.lang.RuntimeException* should be defined for this erroneous situation.

Code/Implementation:

A class implementing the interface specifies how each operation is done and selects a data structure to store a collection of data. For this project, an *array*, **not an array list**, must be used to store the items in an ADT Bag. The element type of the array must be *Object* – the ultimate superclass of all other classes. The items in the array must be always stored **consecutively**. This means shifting is needed to open a space before insertion operations or to close a gap after removal operations.

Javadoc style comments must be written for all members of a class for all classes. Class comments must be included right above the corresponding class headers. Method comments must be included right above the corresponding method headers. Non-Javadoc comments must be included for each block of statements in a method.

Debug/Testing:

Note: It is required to store all input data **in a text file**. It is required to use **decomposition** design technique(Shown in part IV).

To test the ADT bag design, all operations must be tested. 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 for the driver can begin with the following three methods. Create a helper class with following methods included.

```
public class Helper{
    public static void start(){
```
This void method is decomposed into a few tasks:
Create an empty bag.
Fill the bag by calling *create* with a reference to the bag.
Print the bag by calling *display* with a reference to the bag.
…
```
    }

    public static returnTypeOrVoid create(A reference to a bag){
```
Create objects using data from an input file and stores the objects into the bag.
…
Note: Grocery items can be represented as string objects for testing. It is not needed to design a class type for grocery items.
```
    }

    public static returnTypeOrVoid display(A reference to a bag){
```
Displays the objects 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();
        }
    }
```

There are more methods than what are invoked/tested by the previous three methods. You may add more statements in the methods or write additional methods to test the rest of the methods for all classes. 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.

The sample input file may contain items like this:

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

The path to the text file should NOT be a specific folder on your computer. It should start from the default folder. In Eclipse, the default folder is the Java Project folder (the parent folder of *src* folder). When a project is graded, its input file is expected to be in the default folder. **An improper file path will cause *FileNotFoundException* and may result in low credit or 0 credit.**

Documentation:
    Complete all other documents needed.