**COSC1437 Programming Fundamentals II / ITSE2457 Advanced Object-Oriented Programming**

**The submission opens two days before the due day, and closes right at the due time. It is students' responsibility to submit timely and correctly in order to get credits. Students MUST start each project when it is first discussed in class in order to complete it on time. All projects are individual projects.** Each submission must include all required documents. The last submission will be graded. **Project due days may be changed to better align with lectures.**

If one of the following is true, no credits will be given:

- The project is late.
- The class design is missing.
- Wrong files are submitted or part of the source codes is submitted.
- The project has errors.
- The comments are not complete.
- The project is a copy and modification of another student's project. (Both will receive 0.)

**Software Development General Project Rubric:** (Design is worth 20%; the rest is worth 80 %.)

**Analysis:** There will be no credit if the software doesn't meet customer specification at all.

- Does the software meet the exact customer specification?
- Does the software read the exact input data and display the exact output data as they are shown in sample runs?
- Does each class include all corresponding data and functionalities?

**Design:** There will be no credit if the design is missing.

- Is the main method algorithm an efficient solution?
- Does the main method algorithm convert the input data into desired output data efficiently?
- Is the design (a UML class diagram) an efficient solution?
- Is the design created correctly?

**Code:** There will be no credit if there are errors.

- Are there errors in the software?
- Are code conventions and name conventions followed?
- Does the software use the minimum computer resource (computer memory and processing time)?
- Is the software reusable?

**Debug:**

- Are there bugs in the software?

**Documentation:** There will be no credit if comments are not included.

- Class comments must be included in Javadoc format before a class header.
- Method comments must be included in Javadoc format before a method header.
- Javadoc comments must be included before each instance/static variable.
- More inline comments must be included in either single line format or block format inside each method body.
- All comments must be complete in correct format.

A Java source file can contain a class. Each class is made up of the following components:

**Inside a class,**

- package statement
- *import* statements if any
- Javadoc class comments
- a class header and its body. The open { of a class body must be at the end of the header.

**Inside a class body,**

- instance/static variables with Javadoc comments above each variable
- instance/static methods with Javadoc comments above each method header. The open { of a method body must be at the end of the header

**Inside a method body,**

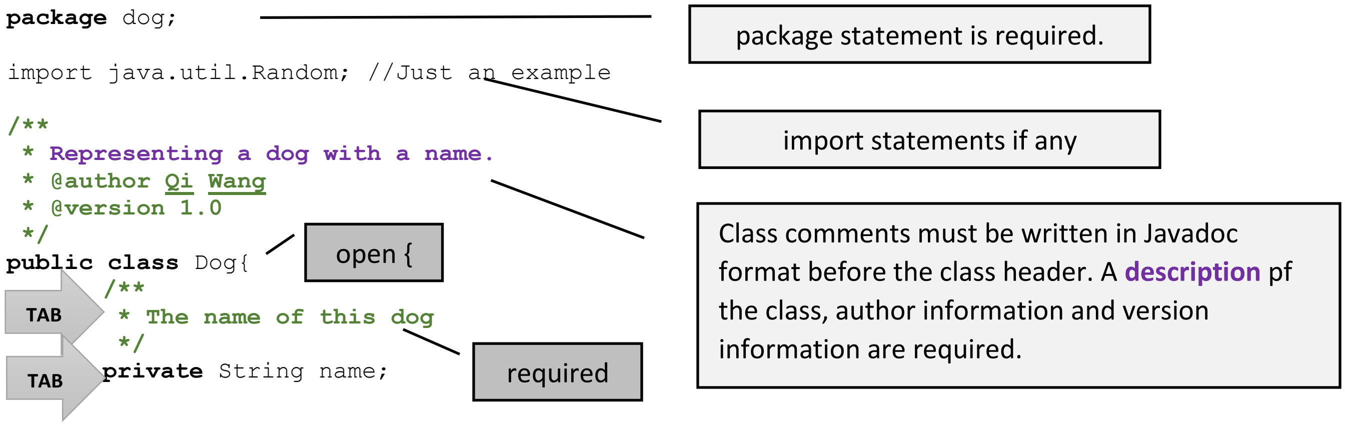
- more comments are included in each method body. These comments can be written in single line or block format.

**Readability:**

To enforce program readability, consistent indentations must be used to show containment hierarchy. Readability is very important. Poor readability of a source file will be ignored or misinterpreted. This will definitely affect a software developer's performance. Please read the following, and careful exam your source file.

1. Align Javadoc comments with its source codes.
2. Use consistent indentations to enhance readability:
   - Indent inside the body of a class for each instance/static variable and method of this class
   - Indent again in each block statement of each method.
3. Use required spaces, not excess spaces to enhance readability:
   - No spaces between comments and the corresponding method header.
   - One space after * at the beginning of each line in Javadoc comments.
   - Align first * of each line in Javadoc comments.
   - One line in between two sections of codes is needed

Comments are required almost everywhere in a source file. Double check to make sure they are completed in the required formats. Code conventions must be used. The following contains additional information on Javadoc. The project description starts on page 4.



```
package dog;

import java.util.Random; //Just an example

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

package statement is required.

import statements if any

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

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

    /**
     * Constructs a newly created Dog object with a name.
     * @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 {

In method comments, the first word must be a **capitalized** verb in the **third** person. Use punctuation marks properly.

Method comments must be written in Javadoc format before the method header. 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.

**Project 5 Exceptions**

**Files to be submitted:**

- A UML design diagram
- Java source files
- Test data file and other supporting files if any

A program can be separated into a normal execution flow and an exception execution flow. A program can handle Java exceptions and/or user-defined exceptions.

**Part I: Normal Execution Flow**

**Geometric object: (*GeometricObject*)**

A geometric object is an object that has a color and a fill. By default, it is white and not filled. Calculating area and calculating perimeter of a geometric object are two common behaviors. Include two abstract methods that will be completed by its sub classes. Complete the rest of the class.

```
private String color;

private boolean filled;

 …

public abstract double getArea();

public abstract double getPerimeter();
```

**Triangle Object: (*Triangle*)**

A triangle is a specific geometric object. Class *Triangle* should be a sub class of *GeometricObject* . By default, a triangle is a default geometric object plus three sides of length 1(*sideOne, sideTwo*, and *sideThree)*. Implement *getArea()* and *getPerimeter().* Complete the rest of the class.

**Part I: Exception Execution Flow**

In a triangle, the sum of any two sides is greater than the third side. Class *Triangle* must adhere to this rule. If a triangle is created with sides that violate the rule, an *IllegalTriangelException* object should be thrown. Create class *IllegalTriangelException* as a subclass of *java.lang.Exception* class. It has a constructor that constructs a new exception with a specified message

```
public class IllegalTriangleException extends Exception  {
      public IllegalTriangleException(String message) { …  }
 }
```

Now it is time to add exception execution flow in *Triangle* class. Check each method, include the codes that may throw exceptions in a *try* block, and then add exception handlers. Always add a generic exception handler at the end to catch any exceptions.

In *Triangle* class, if a method forms a triangle or changes any side of a triangle, this method may throw an illegal triangle exception. Therefore, illegal triangle exceptions should be handled. An exception can be

- caught and handled where it occurs.
- propagated to be handled in next method in calling hierarchy.

In this project, catch and handle *IllegalTriangleException* and other exceptions where they occur. The following shows one example that handles an exception where it occurs. If three sides don't construct a legal triangle, an *IllegalTriangelException* object is thrown, and it is caught in the first exception handler.

```
  public Triangle(int sideOne, int sideTwo, int sideThree){
      try{
           if a legal triangle can be formed with sideOne, sideTwo, and sideThree
               Assign three sides to form a triangle.
           else
               Throw an IllegalTriangelException object using a throw statement.
               This object is caught and handled by the matching exception handler.
     }
     catch (IllegalTriangelException ex){
         Print stack trace
     }
     catch (Exception ex){
         All other exceptions handler
     }
     finally{
          Cleanup code if any
     }
}
```

**Test Class:**

Note: It is required to store all testing data in a file. No credit will be given if not.
It is required to use decomposition design technique. No credit will be given if not.

To test the design, a list of triangles is needed (*ArrayList<GeometricObject>*). To create a *Triangle* object, we only need to store five data: a color, a fill, sideOne, sideTwo, and sideThree. A triangle is created if the sum of any two sides is greater than the third side. Exceptions are thrown otherwise. Here are sample data:

white true 3 4 5

red false 1 2 3

blue false 2 2 1

…

It is REQUIRED to create a list of 10 triangles. Store them in an array list, and print them. It is not efficient to let *main* to do all. Method *main* should be very small and should be the only method in the class. It should invoke a method (*start*) that is decomposed into more methods (*createList*, *displayList*). Include these methods in a separate class. Every method should be designed as a single-minded method.

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

public class TriangleUtility{
       /**
        * Creates a list of triangles, and prints them in order.
        */
       public static void start(){
              ArrayList<GeometricObject> triangles = new ArrayList<GeometricObject>();
              createList(triangles);
              displayList(triangles);
       }

       /**
```

```
         * Creates a list of triangles.
         * @param triangles A reference to an array list of triangles
         */
        public static void createList(){
               Scanner input = …
               …
        }
        /**
         * Displays a list of triangles.
         * @param sections A reference to an array list of triangles
         */
        public static void displayList(ArrayList< GeometricObject> triangles){ … }
}
```

In method *createList*, *next*, *nextBoolean*, and *nextInt methods* of Scanner class are used. Method *next* can throw any one of the following two Java exceptions:

> *NoSuchElementException* - if input is exhausted
> *IllegalStateException* - if this scanner is closed

Method *nextBoolean* can throw any one of the following three Java exceptions.

> *InputMismatchException* - if the next token does not match the *Integer* regular expression, or is out of range
> *NoSuchElementException* - if input is exhausted
> *IllegalStateException* - if this scanner is closed

Method *nextInt* of can throw any one of the following three Java exceptions.

> *InputMismatchException* - if the next token does not match the *Integer* regular expression, or is out of range
> *NoSuchElementException* - if input is exhausted
> *IllegalStateException* - if this scanner is closed

Method *createList* need to handle all three exceptions. The exceptions handlers can simply print the stack trace.

Notice that it is required that exceptions are caught and handled where they occur in this project. Here is information on propagating an exception. An exception can be thrown, and a method further up the call stack will handle it. Propagating an exception is better when there is not enough information to handle the exception properly by the time the code is written. Or the exception can be handled differently by different users of this class.

To specify a method can throw exceptions, add a *throws* clause to the method header for the method. A *throws* clause consists of reserved *throws* and the data type of the exception. Add

```
throws IllegalTriangleException
```

to the end of the method header to specify that this method can throw an *IllegalTriangleException* object. If a method specifies that it can throw exceptions, a *throw* statement must be added the exception is thrown. A *throw* statement consists of reserved word *throw* and an object of the exception class.

```
throw new IllegalTriangleException(“Illegal Triangle: …”)
```

The following codes show the entire example:

```
public Triangle(int sideOne, int sideTwo, int sideThree) throws IllegalTriangleException{
       if a legal triangle can be formed with sideOne, sideTwo, and sideThree{
              //make a triangle/Assign three sides
       }else{
              //Throw an IllegalTriangelException object using a throw statement.
              //The object is passed to and handled by a method further up the call stack
                 throw new IllegalTriangleException(“Illegal Triangle: …”)
       }
}
```

Constructors and method *toString* are tested so far.  Should other methods be tested? Of course! It is your responsibility to test other methods. Keep everything as simple it shows above when you submit the project.