ICSI201 Introduction to Computer Science

Project 05 by Qi Wang

The following parts are included in this document:

- Part I: General project information
- Part II: Project grading rubric
- Part III: Project description
- Appendix A: An example (How to complete a project from start to finish?)

Proper use of the course materials:

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 each project, every single statement for each source file and every single algorithm or class diagram for each design 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.
    - Not readable, 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:
  - An UML class diagram – created either by Violet UML editor or StarUML
  - Java Source code
    - Shape
    - Rectangle
    - Triangle
    - Circle
    - InvalidTriangleException
    - A test class as a helper for a driver program
    - A driver
  - Supporting file
    - An input file, a text file, containing data that can be used to form shapes.

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*, JohnSmithProject05*
  - Zip the folder. A zipped file will be created.
    - For example*,* a file with name *JohnSmithProject05.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 prior to discussing with the instructor. 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

Note: Programs with errors will result in a low credit or no credit.

The following contains only a list of basic PIs used for evaluation. General software development criteria and overall performance of the project are also considered into the final project evaluation.

Project 5: (100 points) **Performance Indicator (PI)**	**LEVELS OF PERFORMANCE INDICATORS**				**Points Earned**
	UNSATISFACTORY	**DEVELOPING**	**SATISFACTORY**	**EXEMPLARY**	
PI1: (10 points) **UML diagram:** • Shape • Rectangle • Circle • Triangle • InvalidTriangleException	None/Not correct at all. (0)	Visibility, name, and type/parameter type/return type, class relationships present for all classes with issues. (<=5)	Visibility, name, and type/parameter type/return type, class relationships present for all classes with minor issues. (<=8)	Visibility, name, and type/parameter type/return type, class relationships present for all classes with no issues. (<=10)	
	Comments:				
PI2a: (5 points): **Implementation:** **Comments (Javadoc style):** • Comments explain the purpose of each part of the program, not how it is coded. • All tags are included correctly. • Class comments • Method comments • Block comments in *main* (non- Javadoc style)	None/Not correct at all. (0)	Some comments are written properly. (<=2)	Most comments are written properly. (<=4)	All comments are written properly. (<=5)	
	Comments:				
PI2b: (5 points): **Implementation:** • Full-word (descriptive) variables • No excessive variables/statements	None/Not correct at all. (0)	Some requirements are met with issues. (<=2)	Most requirements are met with minor issues. (<=4)	All requirements are met with no issues. (<=5)	

• All local variables are declared at the beginning of *main* • Properly formatted code (indentations) • No errors					
	Comments:				
PI2c: (15 points): **Implementation and test:** • Shape class • No errors	None/Not correct at all. (0)	Some methods present with issues. Some requirements are met. (<=7)	All methods present with minor issues. Most of the requirements are met. (<=11)	All methods present with no issues. All requirements are met. (<=15)	
	Comments:				
PI2d: (15 points): **Implementation and test:** • Rectangle class • No errors	None/Not correct at all. (0)	Some methods present with issues. Some requirements are met. (<=7)	All methods present with minor issues. Most of the requirements are met. (<=11)	All methods present with no issues. All requirements are met. (<=15)	
	Comments:				
PI2e: (15 points): **Implementation and test:** • Circle class • No errors	None/Not correct at all. (0)	Some methods present with issues. Some requirements are met. (<=7)	All methods present with minor issues. Most of the requirements are met. (<=11)	All methods present with no issues. All requirements are met. (<=15)	
	Comments:				
PI2f: (15 points): **Implementation and test:** • Triangle class • No errors	None/Not correct at all. (0)	Some methods present with issues. Some requirements are met. (<=7)	All methods present with minor issues. Most of the requirements are met. (<=11)	All methods present with no issues. All requirements are met. (<=15)	
	Comments:				

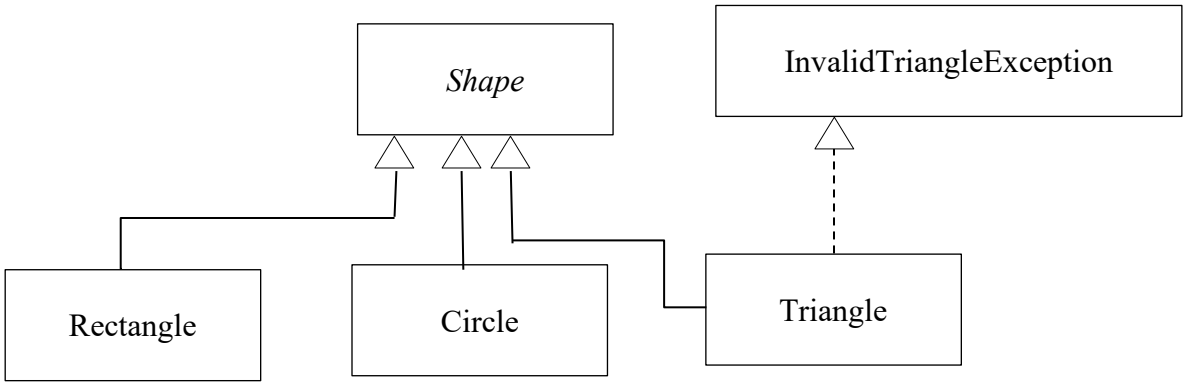
PI2g: (5 points): **Implementation and test:** • InvalidTriangleException class • No errors	None/Not correct at all. (0)	Some methods present with issues. Some requirements are met. (<=2)	All methods present with minor issues. Most of the requirements are met. (<=4)	All methods present with no issues. All requirements are met. (<=5)	
	Comments:				
PI2h: (15 points): **Implementation and test:** • Test class and driver • No errors	None/Not correct at all. (0)	Some methods present with issues. Some requirements are met. (<=7)	All methods present with minor issues. Most of the requirements are met. (<=11)	All methods present with no issues. All requirements are met. (<=15)	
	Comments:				
Overall comments:					
				Total Points Earned	
				Not submitted as a zipped folder	-10
				Total out of 100	

Part III: Project description

Shapes

Write a program to define a generic shape and specific shapes such as a rectangle, a triangle, and a circle. Test each class with its own driver before they can be used. Create a test class (a helper for the driver) to and a driver to test the entire design.

You must complete the design diagram by including the designs of *Shape*, *Rectangle*, *Circle*, *Triangle*, and *InvalidTriangleException* and the class relationships. The *driver* and the *test* class are for testing purpose only. They should not be included in the design diagram.



Shape: An abstract class that defines a Shape object. This class will be tested when its sub classes are tested.

The *Shape* class will have one field:

Fields	**Description**
name	The name of this shape.

The *Shape* class will have the following methods.

Method	**Description**
Default constructor	Creates a Shape instance with a default name.
The second constructor	Given a name, creates a Shape instance with the name.
getName	Retrieves the name of this shape.
setName	Given a name, changes the name of this shape to the new name.
area	Returns the area of this shape. This is an abstract method that will be implemented by the sub classes.
equals	Compares this shape with some other object. This method overrides Java *equals* method.
toString	Represents this shape as a string literal. This method overrides Java *toString* method.

All instance variables and public methods except constructors of a super class are inherited by its sub classes as if they were written in the sub classes.

Rectangle: A sub class of Shape. It defines a *Rectangle* object.

The Rectangle class will inherit all the instance variables from *Shape* class and have the following additional instance variables:

Fields	**Description**
length	The length of this rectangle
width	The width of this rectangle

The *Rectangle* class will inherit all the public methods except the constructors from *Shape* class and have the following methods:

Method	**Description**
Default constructor	Creates a default rectangle instance. By default, a rectangle has the following values for its fields: name: "Rectangle" length: 1.0 width: 1.0
The second constructor	Given a name, a length, and a width, creates a rectangle instance with the name, the length, and the width. For example, new Rectangle ("Small rectangle", 12.3,23.7).
getLength	Returns the length of this rectangle.
setLength	Given a length, changes the length of this rectangle to the new length.
getWidth	Returns the width of this rectangle.
setWidth	Given a width, changes the width of this rectangle to the new width.
area	Returns the area of this rectangle. This method is originally written in the super class. It is implemented to return the area of this rectangle.
equals	Compares this rectangle with some other object. This method overrides the *equals* method from the *Shape* class.
toString	Represents this rectangle as a string literal. This method overrides the *toString* method from the *Shape* class.

Circle: A sub class of *Shape*. It defines a *Circle* object.

The *Circle* class will inherit *name* from the *Shape* class and have the following additional instance variables:

Fields	**Description**
radius	The radius of this circle

The Circle class will inherit all public methods except the constructors from the *Shape* class and have the following additional methods:

Method	**Description**

Default constructor	Creates a default circle instance. By default, a circle has the following values for its fields: name: “Circle” radius: 1.0
The second constructor	Given a name and a radius, creates a circle instance with the name and the radius. For example, new Circle (“Big circle” of radius of 23.7).
getRadius	Returns the radius of this circle.
setRadius	Given a radius, changes the radius of this circle to the new radius.
area	Returns the area of this circle. This method is originally written in *Shape* class. It is implemented to return the area of this circle in this class.
equals	Compares this circle with some other object. This method overrides the *equals* method from the *Shape* class.
toString	Represents this circle as a string literal. This method overrides the *toString* method from the *Shape* class.

Triangle: A sub class of Shape. It defines a *Triangle* object.

The *Triangle* class will inherit *name* from the *Shape* class and have the following additional instance variables:

Fields	**Description**
sideOne	The first side of this triangle
sideTwo	The second side of this triangle
sideThree	The third side of this triangle

The *Triangle* class will inherit all public methods except the constructors from the *Shape* class and have the following methods:

Method	**Description**
Default constructor	Creates a default triangle instance. By default, a triangle has the following values for its fields: name: “Triangle” sideOne: 1.0 sideTwo: 1.0 sideThree: 1.0
The second constructor	Given a name and three sides, creates a triangle instance with the name and the sides. For example, new Triangle (“Right Triangle”,3.0, 4.0, 5.0). 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 *InvalidTriangleException* object with a proper message should be thrown. For example, 4.0, 8.0, and 2.0 can’t form a triangle. * Information on *InvalidTriangleException* is provided later.
getSideOne	Returns the first side of this triangle.
setSideOne	Given a side, changes the first side of this triangle if the new side will form a triangle with the other two sides. Otherwise, an *InvalidTriangleException* object with a proper message should be thrown.

getSideTwo	Returns the second side of this triangle.
setSideTwo	Given a side, changes the side two of this triangle if the new side will form a triangle with the other two sides. Otherwise, an *InvalidTriangleException* object with a proper message should be thrown.
getSideThree	Returns the third side of this triangle.
setSideThree	Given a side, changes the side three of this triangle if the new side will form a triangle with the other two sides. Otherwise, an *InvalidTriangleException* object with a proper message should be thrown.
area	Returns the area of this triangle. This method is originally written in *Shape* class. It is implemented to return the area of this triangle. Heron's formula states that the area of a triangle whose sides have lengths a, b, and c is $A=\sqrt{s(s-a)(s-b)(s-c)}$, where s is the semi perimeter of the triangle; that is, $s=\frac{(a+b+c)}{2}$
equals	Compares this triangle with some other object. This method overrides the *equals* method from the *Shape* class.
toString	Represents this triangle as a string literal. This method overrides the *toString* method from the *Shape* class.

InvalidTriangleException:

Create class *InvalidTriangelException* as a sub class of *java*.*lang*.*Exception* class. In the class, create a non-default constructor that constructs a new exception with a specified message.

```
public class InvalidTriangleException extends Exception{
      public InvalidTriangleException(String message){
            // Call super constructor with the parameter message.
      }
 }
```

This class is used to address an abnormal execution flow in *Triangle* class. In the second constructor and the setters for the sides, 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.

An exception can be caught and handled where it occurs. An exception can be propagated to be handled in next method in calling hierarchy. For this project, catch and handle *InvalidTriangleException* and other exceptions where they occur in *Triangle* class. The following shows one example that handles *InvalidTriangleException*. If three sides do not construct a valid triangle, an *InvalidTriangelException* object is thrown explicitly and is caught in the first exception handler.

```
public Triangle(String name, int sideOne, int sideTwo, int sideThree){
    try{
         if a valid triangle can be formed with sideOne, sideTwo, and sideThree,
             Form this triangle.
         else
             Throw an InvalidTriangelException object using a throw statement.
             This object is caught and handled by a corresponding exception handler.
   }catch (InvalidTriangelException ex){
```

```
            Print stack trace. // Handle InvalidTriangelException.
        }catch (Exception ex){
            Print stack trace. // handle other exceptions.
        }finally{
             Cleanup code if any.
        }
    }
```

Test the entire design:
Now you have tested each class separately. It is time to test them together. To do so, you will test with a list of shapes. Write a *driver* and a *test* class. This *test* class is the helper of the *driver*. The goal is to test everything in the *test* class and keep the *driver* short. Read the prompts in the following methods for more details.

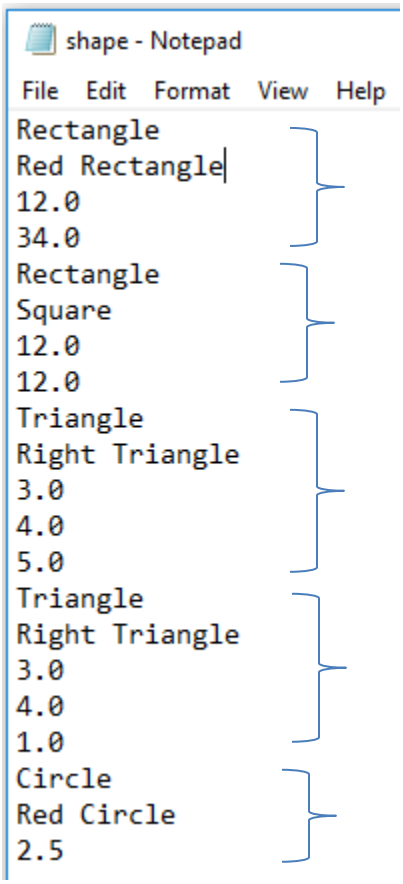
- Create a *test* class with three static methods shown below at minimum.

```
public class Test{
   public static void start(){
      - Create an empty list of shapes and save the reference.
      - Pass the reference to create method that fills the list.
      - Pass the reference to display method that prints the list.

      //The following can be added here or into an additional method.
      - Remove a shape.
      - Display the list again.
      - Check the size of the list.
      - …
   }

   public static returnTypeOrVoid create(a reference to a list) {
      - Create shape objects using data from an input file.
      - Note: A shape can be a rectangle, a circle or a triangle.
      - Add the objects into the list.
   }

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

}
```

- 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){
               Test.start();
         }
}
```

Data file for testing:
See the next page.



The data file contains sets of data needed to make shape objects. Each set of data starts with a type followed by corresponding values needed for the shape. For example, in the first set of data, type Rectangle is included first, the name, the length, and the width are followed. You must add at least ten more sets of data to the file for testing.

Make sure that all methods are tested. You may add more statements in the method *start* or more methods in *Test* class. Notice that the *driver* and the *Test* class are for testing purpose only. They should not be included in the diagram. But you need to submit their source codes.

Test your program with many valid input values. Before you get started, I recommend that you study the example included in appendix A.

Appendix A: An example (How to complete a project from start to finish?)

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

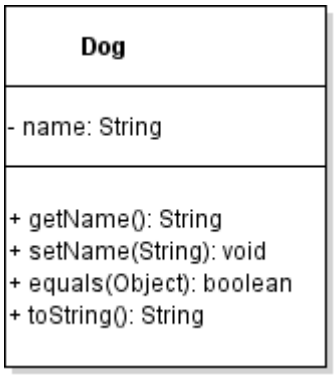
Analysis-design-code-test/debug-documentation.

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 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.

open {

TAB

TAB

open {

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.

There can be many classes in a design. Test each class separately. And then, test the entire design.

- **First, test each class separately.**
  - 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 two 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 the 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, after the following statement, the firstDog's name is stored in firstDogName. The method is called on the right-hand side of the assignment.

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

    You may print the value stored in firstDogName to verify.

```
System.out. println(firstDogName);
```

  - If a method is a void method, simply invoke the method. Use other method to verify its impact. For example, 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.

```
System.out.println(secondDog.getName());
```

  - Repeat until all methods are tested.

- **Next, test the entire design.**
  To do so, you will test with a list of dogs. Write a *driver* and a *test* class. This *test* class is the helper of the *driver*. The goal is to test everything in the *test* class and keep the driver short. Read the prompts in the following methods for more details.
  - Create a *test* class with three static methods shown below at minimum.

```
public class Test{
   public static void start(){
      - Create an empty list of dogs and save the reference.
      - Pass the reference to create method that fills the list.
      - Pass the reference to display method that prints the list.

      //The following can be added here or into an additional method.
      - Remove a dog.
      - Display the list again.
      - Check the size of the list.
      - …
   }

   public static returnTypeOrVoid create(a reference to a list) {
      - Create dog objects using data from an input file.
      - Add the objects into the list.
   }

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

```
                                …
        }

    }
```

- 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){
         Test.start();
   }
}
```

Make sure that all methods are tested. You may add more statements in the method *start* or more methods in *Test* class. Notice that the driver and the *Test* class are for testing purpose only. They should not be included in the diagram. But you need to submit their source codes.