CSCI165 Computer Science II
Lab Exercise
Beginning Object Orientation

Objectives

- Define custom classes
- Create UML diagrams
- Encapsulate data
- Define methods and constructors
- Include appropriate access modifiers and validation
- Install and configure JUnit and pass some provided tests

The Fraction Class

A very common example to show the details of implementing a user-defined class is to construct a class to implement the abstract data type Fraction. We have already seen that Java provides a number of numeric data types, but there are times that it would be most appropriate to be able to create objects that both look and act like fractions.

A fraction such as 3/5 consists of two parts. The top value, known as the numerator, can be any integer. The bottom value, called the denominator, can be any integer greater than 0 (negative fractions have a negative numerator). Although it is possible to create a floating point approximation for any fraction, we would like to represent the fraction using exact values to avoid problems inherent in approximations.

Since defining a class makes a new data type, the operations for the Fraction type will allow a Fraction object to behave like any other numeric type. We need to be able to add, subtract, multiply, and divide fractions. We also want to be able to print fractions using the standard “slash” form, for example 3/5. In addition, all fraction methods should return results in their lowest terms so that no matter what computation is performed, we always end up with the most common form.

The object-oriented principle of **encapsulation** is the notion that we should hide the contents of a class, except what is absolutely necessary to expose. Hence, we will restrict the access to our class as much as we can, so that a user can change the class properties and behaviors only from methods provided by the class. This allows us to embed validation in our objects such that certain criteria cannot be allowed; ie the zero or negative denominator. Java allows us to control access to class features with the **access modifiers** public and private. It is typical in object orientation to make all data attributes private and most methods public, unless there is strong reason otherwise. All attribute variables marked private will only be able to be accessed by the object’s class methods, not directly by the user. Only public methods can be accessed and used by the user. A third access keyword, **protected** will be discussed later.

Keep scrolling for good times . . .

TASK ONE: Using the diagramming tool https://app.diagrams.net/ complete the provided UML diagram for a Fraction class. The diagram should contain

- **Private Encapsulated Data**
  - An integer instance variable for the numerator
  - An integer instance variable for the denominator

- **Public Access Methods**
  - Two constructors. One accepting both the numerator and denominator and one accepting just the numerator.
  - Set Methods for both numerator and denominator.
  - Get Methods for both numerator and denominator.
  - Java expects all classes to have a ***public String toString()*** method.
  - An ***boolean equals(Fraction otherFraction)*** method for equality testing.
  - The method ***int compareTo(Fraction otherFraction)*** described below
  - The method ***double toDecimal()*** that returns the fraction as a decimal number
  - A method called ***void reduce()*** that reduces ***this*** fraction. This method does not return a new Fraction, it simply modifies the numerator and denominator
  - A void method called ***add()*** that accepts a Fraction object, described below.
  - Methods for the remaining arithmetic operations: -, /, *
  - For you folks with C++ experience it is important to note that Java prohibits operator overloading. These behaviors must be implemented as public methods. This is an argument in favor of C++ in my opinion.

- **Private Helper Methods**
  - A method called ***gcd*** that accepts two integers and returns the integer greatest common divisor.

- **CHECK:** You should have a second set of eyes on this before moving on.

TASK TWO: Implement the class design from task one in Java. Finish the ***Fraction*** class in the source code file ***Fraction.java***

Implementation Details:

- Document your method definitions using ***Javadoc*** style comments. Examples are shown in the provided class.
- The constructor that accepts just the numerator should initialize the denominator to 1
- The constructor that accepts the numerator and denominator should prohibit any denominator values being zero.
- The ***setDenominator*** method should prohibit zero.
- The ***reduce*** method should use the ***gcd*** method to help.

- The arithmetic methods are all void. They should accept a Fraction object and apply the operation to ***this*** fraction. “This” refers to the “calling instance”

```
Fraction a = new Fraction(1, 2); // 1/2
Fraction b = new Fraction(3, 4); // 3/4

a.add(b);   // add Fraction b to "this" fraction a
            // add method will modify "this" fraction
            // Result is Fraction "a" becomes 5/4 or 1 1/4 when reduced
```

- The ***compareTo*** method should work in the following way
    - Return 1 if the calling instance is larger than the argument
    - Return 0 if the calling instance equals the argument
    - Return -1 if the calling instance is less than the argument
    - More on the ***Comparable*** design pattern later. This is a simplified introduction

```
Fraction a = new Fraction(1, 2); // 1/2
Fraction b = new Fraction(3, 4); // 3/4

int result = a.compareTo(b);    // compare 1/2 to 3/4
                                // Fraction a is the calling instance "this"
                                // Fraction b is the argument
                                // result should be -1
```

TASK THREE, The Driver: All Java applications must have a single ***public static void main(String[] args)*** method. When building large, multi-class applications, it is beneficial to have this method in it’s own class. From here on out we will be defining a special class to just hold the “main logic”. This class will be referred to as the driver. In labs you will define driver logic to ***prove to the graders*** that your class meets the specifications by creating instances of the class being designed, and calling methods to ensure that they function correctly. Be sure to include descriptive messages in your output. Let the graders know exactly what they are looking at when your code is executed.

For the Fraction class, complete the following driver logic. Print the return of calling ***toString*** on the Fractions to display results. For instance create a Fraction instance using 15 and 25 and then call toString to show that the Fraction is correctly reduced to **3/5**

- Create four Fraction instances. Be sure to demonstrate that your constructors can handle invalid denominators. **Prove it.**
- Call the various instance methods, passing in appropriate data. Display messages that clearly show that the methods are working properly and manipulating the objects as designed.
- Create a 100 element array of Fractions: ***Fraction[] fractions = new Fraction[100];***
- Fill the array with 100 Fraction instances using the data in the file ***fractions.txt***
    - ***Example:*** fractions[index] = new Fraction(numerator, denominator);

- In the driver create a method called ***public static Fraction largestFraction(Fraction[ ] fractions)*** (notice that methods in the Driver are ***static*** while methods in the Fraction class are not static, this won't **always** be the case). This method should accept the array of Fractions and find the largest Fraction in the collection. Use ***compareTo*** for this.

TASK FOUR, Unit Tests:
Your Fraction class must pass all tests in the provided ***FractionTests.java*** file. We will be using the ***Test Runner*** ability of the ***Extension Pack for Java*** VS Code extension to run JUnit tests. See the JUnit tutorial document. We will do an example in lab together on Wednesday.

There will more be 3rd party library installation and configuring. You should have this done before lab on Wednesday

SUBMISSION: Push all relevant files to your ***week6*** repo folder. This should include

- Fraction.java
- Driver.java
- UML image file. Be sure to ***export*** as an image type. Do not submit the raw XML file.

RUBRIC

Requirement	Points
Correctness: code meets all requirements and functions as described in the instructions	15
Descriptive: Driver logic includes sufficiently descriptive messages	5
Javadoc: methods are documented using appropriate Javadoc style comments	5
Unit Tests: tests are complete and robust. All tests pass	5