CSCI 1133
Exercise Set 13

These problems are provided for practice writing simple Java programs. You should complete as many of these problems as you can. Practice is *essential* to learning and reinforcing computational problem solving skills. We encourage you to do these without any outside assistance.

Create a directory named `exercise13` and save each of your problem solutions in this directory. You will need to save your source files in this directory and push them to your repository in order to get help/feedback from the TAs. Name your individual Java source files per the directions for each problem. Automatic testing of your solutions is performed using a GitHub agent, so it is necessary to name your source files precisely and push them to your repository.

The following exercises were completed in earlier Exercise Sets using Python. It is helpful when learning a new programming language to translate programs you have already "solved" into the new language.

These exercises should all be completed using the Java programming language.

A. **Circle Class**

Write the definition for a class named `Circle` that has a *single* private attribute named `radius` representing the radius of the circle, one mutator method: `setRadius(r)` that will update the radius value, and two accessor (getter) methods: `getCircumference()` and `getArea()` that return the circumference and area of the circle, respectively. The class constructor should take a single argument containing the radius Include a main method that will demonstrate your class by instantiating, computing circumference/area and modifying several circles.

Name your main class `Circle` and your source file `Circle.java`

B. **Bug Class**

For this exercise you need to create a class named `Bug` that models a bug moving along a horizontal line. The bug moves either to the right or left. Initially, the bug moves to the right, but it can turn to change its direction. In each move, its position changes by one unit in the *current direction*. The bug class has two integer instance variables: position (0..*n*) and direction (+1: right, -1: left). Implement the following five methods:

- A constructor method with a *single argument* to initialize the initial position of the bug. It should default to the value 0. The starting value of `dir` should always be +1
- A mutator method named `move` with no arguments that will move the bug. The `move` method should not allow the bug to move to a location less than zero!
- A mutator method named `turn` with no arguments that will change the direction the bug is moving
- An accessor method named `display` with no arguments that will display the location and direction of the 'bug'. The display method should output a period ('.') for each position from 0 to the current position of the bug and then output a '>' if the bug direction is to the right or a '<' if the bug is moving to the left. e.g.,:

```
.....<
```

Include a `main` method to first create a `Bug` object at position 10, move the bug, turn the bug, and then (using a loop) move the bug 13 more times. Display the bug's state using the `display` method following each move.

Name your main class `Bug` and your source file `Bug.java`

C. **Stopwatch Class**

Construct a class definition for a `Stopwatch` class that will simulate a simple start-stop timer. Include two private instance variables: `startTime` and `endTime` (floats) and the following methods:

- Constructor method that will initialize both start and end times with the current time of day as returned by the appropriate Java method. The Java `System.currentTimeMillis()` method returns the current time of day in milliseconds.
- Mutator method named `start` that will reset the `startTime` to the current time
- Mutator method named `stop` that will set the `endTime` to the current time
- Accessor method named `elapsedTime` that returns the elapsed time for the stop watch as a float
- Accessor methods `getStartTime` and `getEndTime` that return their respective values

Provide a `main` method that will instantiate and use a `Stopwatch` object to determine and print the amount of time it takes to do some simple task.

Name your main class `Stopwatch` and your source file `Stopwatch.java`

D. **Rational Numbers**

Write the definition for a class named `Rational` that represents rational numbers as the ratio of two *integers* (numerator and denominator). Your class definition should include the following:

- A constructor method that will accept two integer arguments and initialize private instance variables representing the numerator and denominator, respectively. Never allow the denominator value to be zero. If zero is given as an argument, set the denominator to 1.

  The constructor should also maintain the fractional value in its *reduced* form, e.g., the fraction 3/6 should be stored as 1/2
- An overloaded `toString()` accessor method that will return the value of the `Rational` object as a `String` in the form:

  numerator /*denominator*

  If the denominator value is 1, simply show the numerator without the "/*denominator*" part.
- Methods add (+), subtract (-), multiply (*) and divide (/) methods that will return a `Rational` object representing the result of the respective operation.
- A `main` method to test your `Rational` class.

Name your main class `Rational` and your source file `Rational.java`

E. **Currency**

Write the definition for a class named `Currency` that represents currency values as two separate *integers* (dollars and cents). Your class definition should include the following:

- A constructor that will takes two integer arguments and initializes private instance variables representing the whole dollars and cents, respectively. Never allow the cents value to exceed 99. Any *cents* value greater than 99 should be reduced by increasing the whole dollar amount as appropriate. For example, 4 dollars and 108 cents should be represented as 5 dollars and 8 cents.
- An overloaded `toString()` accessor method that will return the value of the `Currency` object as a `String` in the form:

  $*dollars*.*cents*

  A zero currency value should be shown as $0.00. Single digit cents values should be padded with a zero, e.g., $5.08 or $10.00
- Methods add (+) and subtract (-),that will return a `Currency` object representing the result of the respective operation.
- An accessor method named product() that will return a `Currency` object representing the product of a `Currency` amount times a scalar numeric value (`float`).
- A `main` method to test/demonstrate your `Currency` class.

Name your main class `Currency` and your source file `Currency.java`