# CMPSC132 Computer Science II: Java
# Homework 6

**Due: Sunday, March 17th, 2024, 11:59pm**

**Learning Objectives**

- Classes and Objects
- Methods

**Deliverables:**

- 6 Java source files.

**Instructions:**

- Please ONLY submit your Java source file (.java) onto Canvas. Please do not submit the .class file.
- Please add the required *comments* on the top of each of your Java program.
  The comments would contain the following contents (for example):

```
/* Author: Zaihan Yang
 * Due Date: January 21st, 2020
 * Purpose: This Java program outputs several messages onto screen.
 * Credits: I finished this program independently by my own and had no help
 * from other people or resources.
 */
```

**Problem Specifications:**

**Problem 1: The Fraction Class (Fraction.java and Fraction_Test.java) (35')**

**Description**

In mathematics, a simple fraction consists of an integer numerator displayed above a line (or before a slash), and a non-zero integer denominator, displayed below (or after) that line. For example, $\frac{1}{2}$, $\frac{3}{4}$ and $\frac{17}{5}$ are all fractions.

The following UML indicates the definition of a Fraction class, in which:

| **Fraction** |
| --- |
| - numerator: int<br>- denominator: int |
| + Fraction()<br>+ Fraction(num:int, demo:int)<br>+ set(num:int, demo:int): void<br>+ add(other:Fraction): Fraction<br>+ subtract(other:Fraction): Fraction<br>+ multiply(other:Fraction):Fraction<br>+ divide(other:Fraction):Fraction<br>+ lessThan(other:Fraction):boolean<br>+ getReciprocal():Fraction<br>+ toString():String<br>+ toDouble():double |

- -: indicates the data field or method to be private
- +: indicates the data field or method to be public.
- numerator and denominator are the two instance data fields of the class.
- The non-argument constructor will call the argument constructor, and set both the numerator and denominator of a fraction object to be 1.
- The argument constructor will set the numerator and denominator of a fraction object to be specific values.
- The ***set()*** method will set the numerator and denominator of a faction to be specific values.
- The ***add()*** method will add up two fractions, and the result is also a fraction, e.g 1/3 + 1/2 = 5/6
- The ***substract()*** method will subtract the *other* fraction from *this* fraction, and the result is also a fraction, e.g 1/ 2 – 1 /3 = 1/ 6
- The ***multiply()*** method will multiple the *this* fraction by the *other* fraction, and the result is also a fraction, e.g 1/ 2 * 1/ 3 = 1 / 6
- The ***divide()*** method will divide the this fraction by the other fraction, and the result is also a fraction, e.g: 1/2 / 1/3 = 3 / 2
- The **lessThan()** method determines whether this current Fraction is less than the other Fraction number. For example, 2/3 is less than 3/4.

- The ***getReciprocal()*** method will return the reciprocal for a this current Fraction. For example, 3/4 its reciprocal is 4/3.
- The ***toString()*** method will convert the numerator and denominator to their respective string format, and concatenate them by a slash, e.g , $\frac{1}{2}$ will be represented as "1/2".

  **Note***: We often implement the toString() method in each class. The benefit of having such a method is that, when we pass a fraction object to the System.out.println() method, the toString() method will be automatically called to print this fraction.
  - For example:
  - Fraction f1 = new Fraction(3, 4);
  - System.out.println(f1); // this will print 3/4 onto screen.

- The ***toDouble()*** method will convert the fraction to its corresponding floating-point number, e.g, 1/2 is 0.5

**Please also make sure that the numerator and denominator of a valid fraction has no common divisor that is greater than 1.** For example, **4/6 is not a valid fraction number, it should be 2/3 instead**. To do this, you need to first find out the **greatest common divisor** of the numerator and denominator, and divide both the numerator and denominator by it. For example, the greatest common divisor of 4 and 6 is 2, and therefore, we need to divide both 4 and 6 by 2, resulting in the valid fraction be 2/3. You can particularly introduce a method named **getGCD()** within the class definition to do this.

Once you have done defining the Fraction class in Fraction.java source file, please write the client program in another java file *Fraction_test.java*, and test the methods of the fraction class.

**Outputs:**

```
$java Fraction_test
Please enter the numerator and denominator of the first Fraction: 3 4
The first fraction is: 3/4
Its double value is: 0.75

Please enter the numerator and denominator of the second Fraction: 5 6
The second fraction is: 5/6
Its double value is: 0.8333
```

```
3/4 + 5/6 = 19/12
3/4 - 5/6 =  -1/12
3/4 * 5/6 = 5/8
3/4 / 5/6 = 9/10

$java Fraction_test
Please enter the numerator and denominator of the first Fraction: 8 12
This fraction is: 2/3
```

## Problem 2: The MyDate class (MyDate.java and MyDate_test.java) (35')

**Description:**

Given the following UML, write a class called MyDate that represents a date consisting of a year, month, and day, and associated methods.

**You are NOT allowed to use LocalDate or LocalDateTime built-in classes. Please have your own implementation.**

| **MyDate** |
| --- |
| - year: int<br>- month: int<br>- day: int |
| + MyDate()<br>+ MyDate(year:int, month:int, day:int)<br>+ getDay():int<br>+ getMonth():int<br>+ getYear(): int<br>+ addDays(days:int): void<br>+ addWeeks(week: int): void<br>+ daysTo(other:MyDate): int<br>+ daysToEndofYear():int<br>+ isBefore(other:MyDate):boolean<br>+ dateInEnglish(): String<br>+ isLeapYear(): boolean<br>+ toString(): String |

- **MyDate():** this non-argument Constructor will call the argument Constructor, and set the year to be 2000, month to be 1, and day to be 1.
- **MyDate(int, int, int):** this argument Constructor will create an MyDate object, and set the year, month and year to be a specific

value.

- **getYear():** return the year value of this MyDate.
- **getMonth():** return the month value of this MyDate
- **getDay():** return the day value of this MyDate.
- **addDays(int days):** this method will move this MyDate object forward in time by the given number of days.
  For example, if the current date is 2020/2/28, if we add it by 3 days, it will become 2020/3/2
- **addWeeks(int weeks):** this method will move this MyDate object forward in time by the given number of seven-day weeks.
  For example, if the current date is 2020/3/7, if we add it by 2 weeks, it will become 2020/3/21.
  You can actually call the addDays() method, and add it by 14 days.
- **daysTo():** this method returns the number of days that between this MyDate object to the other MyDate object.
  For example, the days from 2020/3/7 to 2020/4/1 is 25.
- **toString():** will return the string representation of this MyDate object. For example: "2020/3/6".
- **daysToEndOfYear():** will return an integer indicating how many days till end of this years. For example, 12/1/2023 is 30 days away from 12/31/2023 (end of year).
- **isBefore(other:MyDate):** is returning whether this current date is before the other date.
- **dateInEnglish:** for 10/19/2023, its corresponding date in English is October 19, 2023. Please return a string in this format.
- **isLeapYear():** this is to judge whether the year of this MyDate object is a leap year or not. Return true if it is a leap year, otherwise return false.
  - Use the following criteria to judge a leap year:
  - If the year is divisible by 100 and at the same time also divisible by 400, it is a leap year.
  - If the year is not divisible by 100, but can be divisible by 4, it is a leap year.
  - All others are not leap year.

Once the definition of the MyDate class is done, write another Java file as MyDate_test.java, to test its methods.

**Outputs:**

```
$java MyDate_test
Please enter the year, month, and day of a MyDate: 2020 3 6
After adding 10 days, MyDate will be 2020 3 16
After adding 3 weeks, MyDate will be 2020 4 7
The days between 2020/3/6 and 2020/4/5 is 19 days.
2020/3/6 is at a leap year.
```

## Problem 3: The Account Class (Account.java and Account_Test.java) (30')

Given the following UML, finish the definition of the Account class in java source file Account.java. One of the data field of the Account class is the MyDate class, which is the class you defined in Problem 2.

- The non-argument() constructor will create an Account object with id to be null, balance and annualInterestRate to be 0, and dateCreated to be null.
- The argument() Constructor will create an Account object with id, balance, annualInterestRate and dateCreated set to specific values.

| **Account** |
| --- |
| - id: String<br>- balance: double<br>- annualInterestRate: double<br>- dateCreated: MyDate |
| + Account()<br>+ Account(id:String, balance:double , rate:double, date:MyDate)<br>+ getID():String<br>+ getBalance():double<br>+ getAnnualInterestRate():double<br>+ getDate():MyDate<br>+ setAnnualInterestRate(rate:double)<br>+ getMonthlyInterestRate():double<br>+ getMonthlyInterest():double<br>+ withdraw(amount:double):void<br>+ deposit(amount:double):void<br>+ transfer(other:Account, amount:double):void<br>+ toString(): String |

- **getID():** return the ID of an account object
- **getBalance():** return the current balance of this account object
- **getDate():** return the Date of this account object
- **setAnnualInterestRate(double):** set the annual interest rate to be specific value
- **getMonthlyInterestRate():** return the monthly interest rate of this account. MonthlyInterestRate is annualInterestRate/12. Please return the MonthlyInterestRate in its percentage form, i.e. 0.045 is 4.5%.
- **getMonthlyInterest():** monthly interest can be calculated as balance * monthlyInterestRate.
- **withdraw(double):** will withdraw a certain amount of money from this account.
- **deposit(double):** will deposit a certain amount of money from this account.
- **transfer(other:Account, amount:double):** will transfer a certain amount of money from this account to the other account.
- **toString():** will concatenate the ID, annualInterestRate and current balance to a String. Note*: display the annualInterestRate in its percentage form.

Once the definition for the Account class is done, write a client program as Account_test.java, in which you need to the following things:

- Create one account object *account1* with ID of 1122, a balance of $20,000, an annual interest rate of 4.5%, and the date of the day when you create this object.
- Create another account object *account2* with ID of 2233, a balance of $1,000, and an annual interest rate of 4.5%, and the date of the day when you create this object.
- Display for the above two account objects their ID, balance, annuallyInterestRate , monthlyInterestRate and the date created.
- Suppose we construct transactions on *account1* for successive 5 months, and in each month, you will:

    - + Withdraw an arbitrary amount of money within the range of [0, 2000] from *account1*. Display an error message if no more money can be withdrawn from this account.
    - + Deposit an arbitrary amount of money within the range of [0, 5000] to *account1*.
    - + Display at the end of each month for *account1* its balance and monthlyInterest.
- Based upon the balance of *account1* after the previous step, Transfer $2000 from *account1* to *account2*, if there is enough money in account1 to make the successful transaction. Display the balance of account1 and accoun2 after the transaction.

**Outputs:**

| ID | Balance | AnnualyInterestRate | MonthlyInterestRate | Date |
|---|---|---|---|---|
| 1122 | 20000 | 4.5% | 0.375% | 2020/3/10 |
| 2233 | 1000 | 4.5% | 0.375% | 2020/3/10 |

| Month | ID | Balance | MonthlyInterest |
|---|---|---|---|
| 1 | 1122 | 19500 | 73.125 |
| 2 | 1122 | 23265 | 872.4375 |
| 3 | 1122 | 18052 | 676.95 |
| 4 | 1122 | 16000 | 600 |
| 5 | 1122 | 20000 | 750 |

| ID | Balance |
|---|---|
| 1122 | 18000 |
| 2233 | 3000 |