CS1112 Exercise 13 Name: _____ NetID: _____

You have until *Tuesday, 11/22, at 9pm* to submit problems 1, 2, and 3 on MATLAB Grader. You do not need to get anything checked off for this exercise.

1 Class `Fraction`

Download the file `Fraction.m` from the *Exercises* page; it is an incomplete class definition. Read it, experiment with it, and implement the incomplete methods. Here're the specific things to note and do:

1.1 Read the class comment carefully. In our simple `Fraction` class we simply assume that the numerator and denominator are integers—we do not check for this. A `Fraction` does not need to be in the reduced form, i.e., 16/6 is fine and does not need to be reduced to 8/3. A negative fraction should have the negative sign associated with the numerator, not denominator. This and other requirements of our `Fraction` are taken care of already in the constructor. Read it carefully.

1.2 Read the given method `isLessThan` in the classdef. Do you understand it? Now experiment in the Command Window!

```
a= Fraction(3,4)
b= Fraction(3,6)
a.isLessThan(b)    % True or false? _______
b.isLessThan(a)    % True or false? _______
```

1.3 Complete method `isEqualTo`. Save the file; then create some `Fraction`s and call the `isEqualTo` method in the Command Window for testing! For example,

```
a= Fraction(3,4)
b= Fraction(3,6)
c= Fraction(1,2)
a.isEqualTo(b)    % True or false? _______
b.isEqualTo(c)    % True or false? _______
```

1.4 Complete method `add`. Again there is no need to reduce the fraction. Next try these statements in the Command Window:

```
a= Fraction(3,4)
b= Fraction(3,6)
c= a.add(b)       % What is fraction c?  Is it correct?
```

1.5 Complete method `toDouble` and then try these statements in the Command Window:

```
a= Fraction(3,4)
x= a.toDouble()       % Call a's toDouble method.  Should be 0.75
```

Copy the contents of your completed file `Fraction.m` into the code box for Problem 1 in MATLAB Grader. Test (and correct if necessary) your class definition. Observe from this problem that every method that is newly implemented gets tested. Furthermore, multiple test cases are selected to test each method comprehensively—*each test targets a different scenario of input* in order to determine a method's correctness.

You will not submit anything for 1.6 and 1.7. These problems are just for extra practice with objects and classes.

1.6 A method called `reduce` is written (it should hopefully look familiar—it is the Euclidean algorithm applied to reducing a fraction), it would be nice to call `reduce` whenever we create a `Fraction`! *Back in the full* MATLAB *environment*, read the constructor again and now *un*comment the last statement so that method `reduce` is called whenever a `Fraction` is created. Next test the updated constructor using the following code in the Command Window:

```
a= Fraction(8,6)  % Fraction has the numerator 4 and denominator 3
```

Do not uncomment the call to method `reduce` in the constructor in your submission on MATLAB Grader! For the purpose of testing the other methods in MATLAB Grader, method `reduce` should *not* be called in the constructor.

1.7 You can uncomment the `disp` method in order to display a `Fraction` in the format *numerator/denominator* if you like. This is not required.

2 A dice game using class `Die`

Download and read the files `Die.m` and `diceGame.m`. Notice that a `Die` has `private` properties `top` and `sides`. `public` getter methods are provided in the class definition. Next consider the function `diceGame`; it contains two errors that you need to correct. Start by calling `diceGame` with a small number of trials. Read both the error message *and* the code in `diceGame`! Correct the function so that it behaves as specified.

> Copy the body of your function `diceGame` into the code box for Problem 2 in MATLAB Grader. Test (and correct if necessary) your function.

3 More on class `LocalWeather`

Download file `LocalWeather.m` and read it. Ask questions if there are parts from what we did in lecture (constructor, method `showMonthData`) that you do not understand. Two of the methods, `getAnnualPrecip` and `getMonthlyAveTemps`, are incomplete (contains "dummy code" that does no calculation and only assigns a value to the return parameter); you will complete them later *using the full* MATLAB *environment* first.

3.1 Experiment! First, download the file `ithacaWeather.txt` which contains weather data for the City of Ithaca. In the *Command Window*, instantiate (create) a `LocalWeather` object using the data file:

```
ithaca= LocalWeather('ithacaWeather.txt')
```

You should see the properties of `ithaca` displayed. Note that one of the properties, `temps`, is an *array of* `Interval` *objects*. Type the following commands in the *Command Window*; make sure you understand the syntax for accessing values.

```
disp(ithaca.city)        % display the value in the property city

disp(ithaca.precip)      % display the values in the property precip--a vector!

disp(ithaca.precip(11))  % What is displayed?  What is it? _____________________________

disp(ithaca.temps)       % Matlab says it's a 1-by-12 array of INTERVALs

disp(ithaca.temps(11))   % Notice that the disp method in class Interval is
                         % used to show the data using Interval notation.

disp(ithaca.temps(11).left)  % What is displayed?  What is it? _________________________
```

3.2 Implement function `getAnnualPrecip` which calculates and returns the total annual precipitation. If any month's precipitation data is missing, the returned value should be `NaN`, a value in MATLAB of type `double` that indicates that a value is not-a-number.

Test your updated class. Save class `LocalWeather`, and type the following in the *Command Window*:

```
ithaca= LocalWeather('ithacaWeather.txt')  % instantiate object
% Which of the following two method calls is correct?  Try them!
a = getAnnualPrecip()
b = ithaca.getAnnualPrecip()
```

Change some values in the data file `ithacaWeather.txt` and call your method again. Test your work thoroughly.

3.3 Implement function `getMonthlyAveTemps` which returns the vector (length 12) of monthly average temperatures. Calculate a month's average temperature as the average between the month's high and low temperatures. See the function comment of `getMonthlyAveTemps` for how any missing temperature (`NaN`) should be handled. The built-in function `isnan` can be used to check whether a variable stores the value `NaN`: `isnan(x)` returns `true` if `x` is `NaN` and `false` otherwise.

Again, save and test your updated class. Back in the *Command Window*, call the instance method `getMonthlyAveTemps` to make sure that it works. Then change some data in the data file and test your method again.

> Copy the contents of your completed file `LocalWeather.m` into the code box for Problem 3 in MATLAB Grader. Test (and correct if necessary) your class definition.