DM575: Exercises and Labs

Set 4

1 Fractions

Define a class `Fraction` whose objects represent fractions. Decide which attributes it should have, and which getters and setters should be available for these attributes. The class should also provide the following methods:

- constructors with zero, one or two arguments, building the fraction 1, a whole number or an arbitrary fraction, respectively;
- methods `Fraction add(Fraction f)`, `Fraction subtract(Fraction f)`, `Fraction multiply(Fraction f)` and `Fraction divide(Fraction f)` returning the result of adding, subtracting, multiplying or dividing this fraction with/by fraction `f`, respectively;
- method `void simplify()` that transforms this fraction into an equivalent irreducible fraction (i.e., one where the numerator and denominator do not have common divisors);
- a method `double value()` that returns a floating point approximation of the value represented by this fraction;
- methods `int integerPart()` and `Fraction properPart()` returning the integer and proper part of the fraction (for example, $\frac{8}{3} = 2 + \frac{2}{3}$, where 2 is its integer part and $\frac{2}{3}$ is its proper part);
- a method `boolean equals(Object other)` that checks whether this fraction is equal to other (remember to implement also `int hashCode()` and guarantee that if `a.equals(b)` returns `true` then `a.hashCode()` and `b.hashCode()` return the same value);
- a method `Fraction clone()` that returns a fraction equal to this one;
- a method `String toString()` that returns a textual representation of this fraction.

2 Geometry

A point on a at surface (such as a computer screen) is de
ned by two coordinates, also called its horizontal and vertical components.

1. Define a class `Point2D` whose objects represent two-dimensional points. Decide which attributes this class should have and which getters and setters should be available for these attributes. The class should also provide the following methods:
   - a constructor that creates a point given its coordinates;
   - a method `boolean isOrigin()` testing whether this point is the origin;
   - a method `void move(double deltaX, double deltaY)` that moves this point according to the vector (`deltaX`,`deltaY`);
   - a method `double distanceToOrigin()` that returns this point's distance to the origin;
   - a method `double distanceTo(Point2D point)` that returns this point's distance to point;
   - a method `int howManyOrigins()` that returns the number of objects currently pointing to the origin;
   - a method `boolean equals(Object other)` that checks whether this point is the same as other (remember `int hashCode()`);
   - a method `Point2D clone()` that returns a copy of this point;
   - a method `String toString()` that returns a textual representation of this point.
2. A polygon is a region on the plane limited by straight line segments (its sides).

   Implement a class `Polygon` whose objects represent polygons. A polygon is to be represented as a sequence of points (its vertices) such that there is a line between each two consecutive points, as well as between the rest and the last. Exploit class `Point2D` as much as possible. Decide which attributes this class should have and which getters and setters should be available for these attributes. Each polygon should also have a unique identifier. The class should also provide the following methods:

- a constructor that creates a polygon from an array of `Point2D` containing its vertices;
- a method `double perimeter()` returning this polygon perimeter;
- a method `Point2D nearest()` that returns the vertex of this polygon that is closest to the origin;
- a method `double longestSide()` returning the length of this polygon longest side;
- a method `void move(double deltaX, double deltaY)` that moves this polygon according to the vector (`deltaX`,`deltaY`);
- a method `int verticesInQuadrant(int n)` counting how many of this polygon vertices lie on the `n`-th quadrant;
- a method `boolean isTriangle()` that determines whether this polygon is a triangle;
- a method `boolean isRectangle()` that determines whether this polygon is a rectangle;
- a method `int id()` returning this polygon identifier;
- a method `Polygon mostRecentTriangle()` returning the triangle most recently created;
- a method `boolean equals(Object other)` that checks whether this polygon is equal to other (note that the polygon's vertices need not be given in the same order);
- a method `Polygon clone()` that returns a copy of this polygon;
- a method `String toString()` that returns a textual representation of this polygon.

Make UML diagrams and write clients to test your classes.

3 Time Management

A team developing a project realized that they needed to be able to represent points in time.

1. Implement a class `TimeStamp` whose objects are points in time represented by hours, minutes and seconds. Make a UML class diagram for this class. Decide which attributes this class should have and which getters and setters should be available for these attributes. The class should also provide the following methods:

   - constructors with 0, 1, 2 or 3 arguments, corresponding (respectively) to the hours, minutes and seconds of the required timestamp (assume the default value 0 for absent arguments);
   - a method boolean `valid(int hours, int minutes, int seconds)` that checks whether its given arguments can be passed along to the constructor;
   - methods `void skipSecond()`, `void skipMinute()` and `void skipHour()` that add one second, one minute and one hour, respectively, to this timestamp (assume that `23:59:59` is followed by `0:00:00`);
   - a method `void skipTime(TimeStamp time)` that adds the amount of time described in time to this timestamp;
   - a method `boolean equals(Object other)` that determines whether this timestamp is the same as other (remember `int hashCode()`);
   - a method `TimeStamp clone()` that returns a copy of this timestamp;
   - a method `String toString()` that returns a textual representation of this timestamp.

   Write a client to test this class.

2. As the project continued to grow, the team concluded that in some cases they needed to enrich timestamps with information about the date, represented as a year, month and day. Implement a class Date that represents a timestamp in a particular date, including information about the year, month, day and timestamp. Make a UML class diagram for this class (including `TimeStamp`). Decide which attributes this class should have and which getters and setters should be available for these attributes. Exploit the class `TimeStamp` as much as possible. Your class should provide the same methods as class `TimeStamp` (except the construtors) plus the following ones:

   - a constructor with three arguments that creates an object corresponding to midnight on the given date;
   - a constructor with four arguments that creates an object corresponding to the given timestamp on the given date;
   - a method `boolean valid(int year, int month, int day)` that checks whether its arguments can be passed along to the constructor;

- methods `void skipDay()`, `void skipMonth()` and `void skipYear()` that skip this date forward by one day, one month or one year, respectively;
- a method `int largestYear()` that returns the year of the date most in the future ever created or referenced;
- a method `boolean equals(Object other)` that determines whether this date is the same as other;
- a method `Date clone()` that returns a copy of this date;
- a method `String toString()` that returns a textual representation of this date.

Write a client to test your class.

3. Define an interface `Time` containing the methods for time manipulation common to `TimeStamp` and `Date` and have both classes implement it. Update your UML diagram accordingly.

4. The team later realised that they need to use points in time to sort events. Modify your implementations of `TimeStamp` and `Date` to implement Java `Comparable` interface and update your UML diagram accordingly. Remember to guarantee the requirements by the contract of `compareTo`[1] (`sgn` stands for the mathematical signum, see e.g., `Integer.signum`[2]):

   - `sgn(a.compareTo(b)) == -sgn(b.compareTo(a))`;
   - `a.compareTo(b) > 0 & b.compareTo(c) > 0` implies `a.compareTo(c) > 0`;
   - `a.compareTo(b) == 0` implies `sgn(a.compareTo(c)) == sgn(b.compareTo(c))`;
   - `a.compareTo(b) == 0` iff `a.equals(b)` (this is not a strict requirement but it is strongly recommended).

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html#compareTo-T-
[2] https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#signum-int-