Week 3
Writing Basic Classes in Java

Comment

The purpose of this prac is to provide practice with writing classes, constructors, and dynamic methods, as well writing and understanding static methods, overriding the `toString` method, and generally getting experience with Java syntax.

From this prac forward, whenever we ask you to write a class, you should also write a simple `main` method which tests its functionality. This does not need to be in the same class (not every class needs a `main` - having `class Foo` and `class Bar` being tested by `class Test` is fine), however it can be if you wish.

1 Point

Write a class called `Point` which represents a point on a Cartesian plane. A `Point` stores two `float`s, one representing the x-coordinate, the other representing the y-coordinate. The class should also have the following methods:

1. A constructor which takes two `float`s and stores them as the x- and y-coordinates respectively.
2. A default constructor which stores `0` as both the x- and y-coordinate.
3. `float getX()` and `float getY()` which return the x- and y- coordinate being stored respectively.
4. `Point movePoint(float deltaX, float deltaY)` which returns a new `Point` object. The new `Point`'s x-coordinate should be `this.x + deltaX`, and its y-coordinate should be `this.y + deltaY`. *Note that this method should not modify the Point it is being called on.*

2 Line

Write a class called `Line`. It represents a single line on the Cartesian plane, and stores two `Point` objects (its start and end points), as well as a double representing the length of the line. The class should also have the following methods:

1. A static method `double lineLength(Point start, Point end)` which returns the distance between the two given points (using the mathematical formula $\sqrt{(x2-x1)^2+(y2-y1)^2}$).
2. A default constructor which has both its start and end at the origin (`0, 0`), and has a length of `0`.
3. A constructor which takes two `Point` objects and stores them as the start and end points. It should also store the distance between these two `Point`s.
4. `Point getStart()` and `Point getEnd()` which return the start and end `Point`s of the `Line`, and `double getLength()` which returns its length.

5. `void moveStart(float deltaX, float deltaY)` which moves the `Line`'s start `Point` by the given values. Also write `void moveEnd(float deltaX, float deltaY)` which does the same for the end `Point`. *(Hint: you may find the `Point.movePoint` method useful).*

3 Additional Methods

Add the following methods to the `Point` class:

1. `Line createLine(Point end)` which returns a `Line` object with `this` as the start and `end` as the end.
2. `Point flipPoint()` which returns a new object that is the 'flipped' version of `this Point` (that is, the sign of both its x- and y-coordinate should be changed → $(1, -5)$ becomes $(-1, 5)$).

Add the following methods to the `Line` class:

1. `Point middle()` which returns the mid-point of the start and end of the line (that is, the point at $(\frac{x1 + x2}{2}, \frac{y1 + y2}{2})$).
2. `Line flipLine()` which returns a new object that is the 'flipped' version of `this Line` (that is, both its start and end point should be 'flipped' as described in item 2 above.

4 toString()

1. Override `toString()` in `Point` to return a `String` in the format `(x, y)`, where `x` and `y` are the coordinates of the `Point`.
2. While this particular string is not particularly expensive to calculate, some strings can be time-intensive to compute. Because of this, different techniques are often used to implement the `toString` method which reduce the number of times these expensive operations need to be performed. This is particularly useful in immutable classes, as the `toString` representation is unlikely to change. Implement the `toString` method again in the following ways:

Precalculation	`toString` gets the string to return from a `String` member variable, which is populated in the constructor
Caching	`toString` calculates the result, but only the first time it is called. It stores that result for future calls in a member variable (how do you know whether this is the first call?).