`Point` class extra credit opportunity

This is a non-trivial coding task, worth up to 20 points (even more, if you complete some of the extended specifications portion), that exercises a number of concepts and techniques that we've covered in our previous coding exercises, as well as some that have been discussed in *Java: A Beginner's Guide* and *Effective Java*.

Participating

After reading the specifications, if you want to attempt the problem, please click here (https://classroom.github.com/a/kEMNojs1). As you work on the problem, commit and push as usual. This is already a Git repository, with a remote on GitHub; **there is no need** to re-share on GitHub.

Specifications

For the full 20 points of credit, you must complete the implementation of the `edu.cnm.deepdive.geometry.Point` class. Instances of the class represent points in 2-dimensional space, using Cartesian coordinates; that is, every point has an X and a Y coordinate—both set when the instance is initialized, and accessible via getter methods. Additionally, the class will provide a number of methods that return new `Point` instances, based on common operations.

General characteristics

- This new class must have **no** connection (implicit or explicit) to `java.awt.geom.Point2D` or `java.awt.Point`.
- The class must not be extendable—that is, it must not be possible to create another class that extends this `Point` class.
- Instances of the class must be immutable. (See *Effective Java* for the implications of this on how the class should be declared and implemented.)
- Any member fields, methods, method parameters, and constructor parameters must be named/cased according to the conventions dictated by the Google Java Style Guide. (Don't forget: It's not just an IntelliJ plug-in; there's also a web page you can consult, which details these conventions.)

Creating instances

The class must have no `public` constructors, but must instead implement the following `public static` factory methods:

- `public static Point fromPoint(Point point)`

  Creates and returns an instance as a copy of the specified point.

- `public static Point fromXY(double x, double y)`

  Creates and returns an instance with the specified *X* and *Y* coordinate values.

- `public static Point fromPolar(double r, double theta)`

  Creates and returns a new instance, with *X* and *Y* coordinates based on the following conversions from `r` and θ (`theta`):

  - $x = r \cos(\theta)$
  - $y = r \sin(\theta)$

  See Figure 1 for an illustration of the relationship between Cartesian and polar coordinates.

Hint: Static factory methods may invoke constructors; just because the class has no `public` constructors, that doesn't mean it can't have `private` constructors.

Accessing the instance state

- The class must provide the following basic accessors (getters):

  - `public double getX()`

    Returns the *X* coordinate of the point.

  - `public double getY()`

    Returns the *Y* coordinate of the point.

- The class must provide the following convenience methods:

  - `public double[] getCoordinates()`

    Returns the *X* and *Y* coordinates as an array: the *X* value should be in element 0 of the array, while the *Y* value should be in element 1.

  - `public double getR()`

    Computes and returns the distance of the instance from the origin, using the familiar Pythagorean theorem computation:

    $$r = \sqrt{x^2 + y^2}$$

(Hint: See the `java.lang.Math` class for a method that makes this easy.)

- `public double getTheta()`

  Computes and returns the angle formed between the positive *X*-axis and the line segment drawn from origin to this point instance (see Figure 1). That angle is measured in counter-clockwise radians. Fundamentally, this is given by the formula:

$$\theta = \tan^{-1}(y/x)$$

  However, this would (for example) give the same angle for a point at (1, 1) as it does for a point at (-1, -1). It also has problems for any point with an *X* value of 0. So you will need to address this accordingly. (Hint: Review the `java.lang.Math` class to find a method that returns an angle, based on separate *X* and *Y* values, rather than on the quotient of the two.)

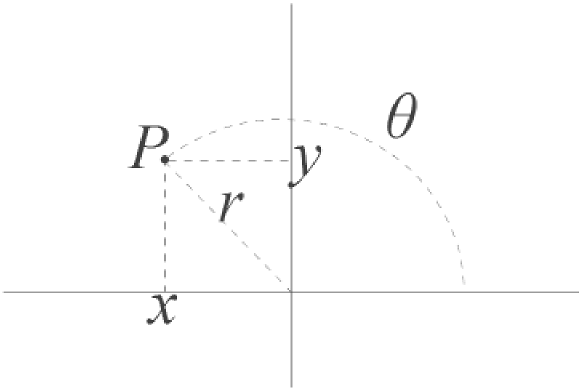

Figure 1: Cartesian and polar coordinates.

Overriding `java.lang.Object` methods

Your implementation should override the following methods inherited from `Object`:

- `public boolean equals(Object other)`

  Must return `true` if `other` is actually an instance of `Point`, and if its coordinate values are equal to the coordinates of this instance.

- `public int hashCode()`

  Computes and returns a hash code constructed from the field values of this instance. Since `Point` is designed to be immutable, this must return the same value every time it is invoked on a given instance. (Hint: Review the `java.util.Objects` class for methods that might help you compute a hash code.)

- `public String toString()`

  Computes and returns a string representation of this `Point` instance. This should take the form `Point({x}, {y})`, with `{x}` and `{y}` replaced by the corresponding coordinate values. For example if `p` is a `Point` instance with an *X* coordinate value of 1.0, and a *Y* coordinate value of 2.0, `p.toString()` should return `"Point(1.0, 2.0)"`. (Hint: Use the `String.format` method for constructing a formatted `String`.)

Constants

The class must have the following `public static final` (constant) field, of the `Point` type, and initialized accordingly:

- `public static final Point ORIGIN`

  Represents the point at (*X*, *Y*) coordinates (0, 0)—that is, the origin of the Cartesian coordinate system. (Hint: You can call a method or create an instance with a constructor to assign a value to this field.)

Geometric computations

The class must implement the following methods, to perform the described operations:

- `public Point add(Point other)`

  Creates and returns a new `Point` instance, with *X* and *Y* coordinates computed from the sums of the respective coordinates of this instance and `other`. That is, the *X* coordinate of the new instance should be equal to `this.x + other.x`, while the *Y* coordinate should be equal to `this.y + other.y`. (Of course, this assumes that the fields storing the coordinates will be called `x` and `y`; that detail is left up to you.)

- `public Point multiply(double scale)`

  Computes and returns a new `Point` instance, with *X* and *Y* coordinates computed from the product of the respective coordinates of this instance and `scale`. That is, the *X* coordinate of the new instance will be computed as `scale * this.x`, while the *Y* coordinate will be computed as `scale * this.y`. (Again, this assumes that the fields storing the coordinates are called `x` and `y`.)

Extended specifications for additional points

For additional points (up to 20 beyond the base 20), implement some portion of the items below.

Additional geometric computations

- `public Point subtract(Point other)`

  Equivalent to `add(other.multiply(-1))`

- `public Point divide(double scale)`

  Equivalent to `multiply(1 / scale)`

- `public double dot(Point other)`

  Computes and returns the *dot product* of this instance and `other`. The dot product is simply the sum of the coordinate products—that is, `this.x * other.x + this.y * other.y`.

`java.lang.Comparable` implementation

Implement *`Comparable`* `<Point>`—which implies implementing `public int compareTo(Point other)`—to support comparing this instance with another, based on each instance's distance from the origin. That is, if `this` is closer to the origin than `other`, then `this.compareTo(other)` must return a negative value; if `other` is closer to the origin than `this`, `this.compareTo(other)` must return a positive value; if the distance to the origin is the same for `this` and `other`, `this.compareTo(other)` must return zero.

`java.util.Comparator` implementations

- Define a `public static class XYComparator`, within `Point`, that implements *`Comparator`* `<Point>`. This implementation should implement the `public int compare(Point p1, Point p2)` method to compare first on the two instance's *X* coordinate values, and then (if the *X* values are equal) on their *Y* coordinate values.

- Define a `public static class YXComparator`, within `Point`, that implements *`Comparator`* `<Point>`. This implementation should compare first on the two instance's *Y* coordinate values, and then (if the *Y* values are equal) on their *X* coordinate values.

- Define a `public static class ManhattanComparator`, within `Point`, that implements *`Comparator`* `<Point>`. This implementation should compare the two instances based on their *Manhattan* (rectilinear) distances from the origin. For example, a point at (-2, 2) has a Manhattan distance from the origin (i.e. the sum of the absolute values of its coordinates) of 4; a point at (3, 0) has a Manhattan distance from the origin of 3; your `ManhattanComparator` implementation should return a positive value when comparing the first to the second (in that order), since the first has a greater Manhattan distance.

Alternative/additional `Comparator` implementations

Instead of (or in addition to—see below) creating the 3 *`Comparator`* `<Point>` implementations above, create 3 `public static final Comparator<Point>` fields, with the following names:

- `XY_COMPARATOR`
- `YX_COMPARATOR`
- `MANHATTAN_COMPARATOR`

These may be instances of the `java.util.Comparator` implementations specified above, or they may be written as lambdas or anonymous classes; if the latter, then you may leave out the actual class definitions specified in "`java.util.Comparator` implementations".