Programmazione ad Oggetti mod. 2

6/9/2019

Studente _____ Matricola _____

1. Si implementi in Java 8+ un sistema di classi che rappresentano punti, linee e poligoni regolarli nel piano cartesiano $R \times R$.
   (a) [1 punti] Si implementi un tipo che rappresenta punti bidimensionali immutabili, ovvero una classe `Point` in cui le componenti `x` ed `y` sono di tipo `double`.
   (b) [2 punti] Si implementi un tipo che rappresenta segmenti bidimensionali immutabili, ovvero una classe `Line` il cui costruttore prende due argomenti di tipo `Point`. Essa deve fornire un metodo `length()` che ne calcola la lunghezza tramite la distanza euclidea tra i due punti.
   (c) Si prenda in considerazione la seguente classe astratta `Polygon` che rappresenta poligoni regolari come liste di punti (minimo 3, verificato a runtime). I punti nella lista determinano l'ordine di costruzione dei segmenti di cui è composto il poligono. Ad esempio, una lista contenente i seguenti 3 punti $A = (0, 0)$, $B = (3, 3)$ e $C = (3, 0)$ rappresenta un triangolo rettangolo in cui il primo lato è AB, il secondo è BC ed il terzo è CA.

```
public abstract class Polygon {
    protected final List<Point> points;
    protected Polygon(List<Point> points) {
        assert points.size() >= 3;
        this.points = points;
    }
    public Iterator<Line> lineIterator() { /* da implementare */ }
    public double perimeter() { /* da implementare */ }
    public abstract double area();
}
```

      i. [2 punti (bonus)] Per quale motivo è necessario vincolare a runtime la dimensione minima della lista di punti tramite un `assert` anziché sfruttare in qualche modo il type system per fare un controllo statico? Si articoli una breve risposta.
      ii. [6 punti] Si implementi il metodo `lineIterator()` che costruisce un iteratore su oggetti di tipo `Line` e si comporta come un *wrapper* dell'iteratore estratto dal campo `points`. Gli oggetti prodotti dall'iteratore di `Line` devono essere costruiti *dinamicamente* leggendo coppie di punti adiacenti dall'iteratore di `Point`. Si implementi una logica di *caching* dell'ultimo punto letto per permettere la costruzione di un nuovo segmento adiacente all'ultimo ad ogni invocazione del metodo `next()`. Si badi inoltre a riusare opportunamente il primo punto come secondo estremo dell'ultimo segmento costruito.
      iii. [2 punti] Si implementi il metodo `perimeter()` in funzione del metodo `lineIterator()`, ovvero calcolando il perimetro del poligono iterando sui segmenti che lo compongono.

2. Estendiamo ora la gerarchia di classi introducendo tipi specializzati per i poligoni classici.
   (a) [4 punti] Si implementi una sottoclasse di `Polygon` di nome `Triangle` che rappresenta triangoli qualunque.

```
public class Triangle extends Polygon {
    public Triangle(Point p1, Point p2, Point p3) { /* da implementare */ }
    @Override
    public double area() { /* da implementare */ }
}
```

Il costruttore prende 3 argomenti di tipo `Point` e deve chiamare il super-costruttore opportunamente. Si implementi il metodo `area()` in modo che calcoli l'area del triangolo senza fare assunzioni sulla sua forma.

(b) [3 punti (bonus)] Si implementi un tipo che rappresenta triangoli rettangoli tramite una sottoclasse di `Triangle`. Si badi in particolar modo a definire un costruttore che sintetizzi al massimo le informazioni passate come argomenti. Si prediligano i controlli statici a quelli dinamici, se possibile, e si tenti di minimizzare i casi di ambiguità o gli stati di invalidità dell'oggetto; nel caso in cui si ritenga necessario fare dei controlli a runtime, si usi il costrutto `assert`.

(c) [4 punti] Si implementi una sottoclasse di `Polygon` di nome `Rectangle` che rappresenta rettangoli.

```
public class Rectangle extends Polygon {
    public Rectangle(Point p1, Point p3) { /* da implementare */ }
    @Override
    public double area() { /* da implementare */ }
}
```

Il costruttore prende i 2 punti della diagonale e deve passare al super-costruttore i 4 punti che costituiscono il rettangolo calcolandone le coordinate per proiezione ortogonale. Si badi all'ordine in cui i punti compaiono nella lista, affinché siano adiacenti e permettano di comporre i lati correttamente. Si implementi il metodo `area()` in modo che calcoli l'area del rettangolo.

(d) [4 punti] Si implementi una sottoclasse di `Rectangle` di nome `Square` che rappresenta quadrati.

```
public static class Square extends Rectangle {
    public Square(Point p1, double side) { /* da implementare */ }
}
```

Il costruttore prende il punto in basso a sinistra e la dimensione del lato e deve chiamare il super-costruttore opportunamente.

3. Si prenda in considerazione il seguente codice, in cui compare l'invocazione di un metodo statico `max` da definire:

```
Square sq1 = new Square(new Point(10., -4.), 0.1),
       sq2 = new Square(new Point(1., 20.), 0.01);
Collection<Square> squares = List.of(sq1, sq2);
Rectangle r = max(squares, new Comparator<Polygon>() {
    @Override
    public int compare(Polygon a, Polygon b) {
        return (int) (a.area() - b.area());
    }
});
```

(a) [5 punti] Si scriva la firma e l'implementazione del metodo statico `max()` invocato nell'ultimo statement affinché il codice di cui sopra compili correttamente. Si renda tale metodo più generico possibile e non monomorfo rispetto ai tipi che compaiono in questa invocazione, prestando particolare attenzione ai vincoli sui generics. La semantica di `max` è facilmente intuibile: trova l'elemento maggiore usando il comparatore per confrontare gli elementi della collection di input.

(b) [½ punti (bonus)] Assumendo il comportamento corretto del metodo `max()`, quale delle seguenti espressioni booleane è vera?

◯ `sq1 == r` ◯ `sq2 == r` ◯ `r == null` ◯ nessuna delle precedenti

(c) [½ punti (bonus)] Quale dei seguenti numeri razionali rappresenta il valore di tipo `double` computato dall'espressione `r.area()`?

◯ 10^{-1} ◯ 10^{-2} ◯ 10^{-4} ◯ non è un quadrato ma un rettangolo

Question:	1	2	3	Total
Points:	11	12	5	28
Bonus Points:	2	3	1	6
Score:				