Programmazione ad Oggetti
Mod. 2

31/1/2020

Studente _____ Matricola _____

Definiamo in Java 8+ un sistema di classi ed interfacce che rappresentano punti e solidi nello spazio $\mathbb{R}^3$. Per semplificare i calcoli legati alle coordinate assumiamo che tali solidi siano sempre orientati ortogonalmente rispetto agli assi cartesiani.

1. 2 punti Si implementi un tipo che rappresenta punti tridimensionali *immutabili* nello spazio, ovvero una classe `Point` con i campi pubblici `x`, `y` e `z` di tipo `double` e l'opportuno costruttore. Si implementi anche un metodo di `Point` avente firma `Point move(double dx, double dy, double dz)` che produce un nuovo punto *traslato* rispetto a `this` di dx, dy e dz rispettivamente per ogni dimensione.

2. Si prendano in considerazione le seguenti interfacce Java:

```java
public interface Solid extends Comparable<Solid> {
    double area();
    double volume();
    PositionedSolid at(Point origin);

    static <S extends Solid> int compareBy(Function<S, Double> f, S s1, S s2) {
        return Double.compare(f.apply(s1), f.apply(s2));
    }
    static <S extends Solid> Comparator<S> comparatorBy(Function<S, Double> f) {
        return (s1, s2) -> compareBy(f, s1, s2);
    }
    default int compareTo(Solid s) {
        return compareBy((x) -> x.volume(), this, s);
    }
}

public interface Polyhedron extends Solid {
    double perimeter();
    @Override
    PositionedPolyhedron at(Point origin);
}

public interface PositionedSolid {
    Point origin();
}

public interface PositionedPolyhedron extends PositionedSolid, Iterable<Point> {}
```

L'interfaccia `Solid` rappresenta oggetti tridimensionali *senza* una posizione specifica nello spazio; l'interfaccia `Polyhedron` rappresenta il sottoinsieme dei poliedri come sottotipi di `Solid` aventi un perimetro ed un numero finito di punti. Per posizionare un solido nello spazio è necessario invocare il metodo `at()`: il tipo `PositionedSolid` rappresenta solidi *con* una certa posizione nello spazio, detta origine. Il tipo `PositionedPolyhedron` specializza `PositionedSolid` aggiungendo ad un poliedro posizionato nello spazio la capacità di essere iterabile; l'iteratore deve fornire i punti di cui è costituito il poliedro, in un ordine qualunque.

(a) [2 punti] Quale caratteristica del linguaggio Java è all'opera nell'override del metodo `at()` all'interno dell'interfaccia `Polyhedron`?

- ◯ E' un overload, non un override.
- ◯ L'overriding supporta la *co-varianza* del tipo di ritorno di un metodo.
- ◯ L'overriding supporta la *contro-varianza* del tipo di ritorno di un metodo.
- ◯ Un override può cambiare liberamente il tipo di ritorno di un metodo.

(b) [2 punti (bonus)] La lambda espressione `(x) -> x.volume()` passata come primo argomento al metodo `compareBy()` nel corpo del metodo `compareTo()`, avente tipo `Function<Solid, Double>`, è equivalente a quale dei seguenti costrutti del linguaggio Java?

- ◯ Ad un riferimento ad un metodo non-statico (*instance method reference*) di un oggetto arbitrario di tipo `Solid`; ovvero all'espressione `Solid::volume`.
- ◯ Ad un riferimento ad un metodo non-statico (*instance method reference*) di un oggetto specifico, che è `this` in questo caso; ovvero all'espressione `this::volume`.
- ◯ Ad un riferimento ad un metodo statico (*static method reference*) della classe `Solid`; ovvero all'espressione `Solid::volume`.
- ◯ A nessuna dei precedenti.

3. [6 punti] Si implementi la classe `Cube`, con il suo costruttore parametrico sulla lunghezza del lato ed i metodi richiesti dalle interfacce implementate.

```
public class Cube implements Polyhedron {
    private double side;    // lato del cubo
    /* implementare il resto */
}
```

Suggerimento: si faccia particolare attenzione all'implementazione dell'iteratore di punti richiesto dall'interfaccia `Iterable<Point>` implementata dalla classe `PositionedPolyhedron`. Si usi il metodo `move()` di `Point` per calcolare *al volo* i vari punti di cui è costituito il cubo. Si consiglia infine di implementare il metodo `at()` tramite una anonymous class.

(a) [1 punti (bonus)] Sarebbe possibile implementare il metodo `at()` tramite una lambda? ◯ si ◯ no

4. [6 punti] Si implementi la classe `Sphere`, con il suo costruttore parametrico sulla lunghezza del raggio ed i metodi richiesti dalle interfacce implementate[1]:

```
public class Sphere implements Solid {
    private double ray;     // raggio della sfera
    /* implementare il resto */
}
```

5. Si prenda ora in considerazione questo *main* che costruisce alcune liste di solidi e poliedri:

```
public static void main(String[] args) {
    Cube cube1 = new Cube(11.), cube2 = new Cube(23.);
    Sphere sphere1 = new Sphere(12.), sphere2 = new Sphere(35.);
    List<Solid> solids = List.of(cube1, cube2, sphere1, sphere2);
    List<Cube> cubes = List.of(cube1, cube2);
    List<Sphere> spheres = List.of(sphere1, sphere2);
    List<? extends Polyhedron> polys = cubes;
}
```

(a) [2 punti] Si ordini in ordine crescente la lista di cubi `cubes` secondo il valore del volume, scrivendo uno statement di invocazione del seguente metodo della classe `Collections` del JDK:

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

(b) [3 punti] Si ordini in ordine crescente la lista di sfere `spheres` secondo il valore della superficie totale, scrivendo uno statement di invocazione del seguente metodo della classe `Collections` del JDK:

[1] Si rammenti che una sfera di raggio r ha volume $V = \frac{4}{3}\pi r^3$ e superficie totale $A = 4\pi r^2$.

```
static <T> void sort(List<T> list, Comparator<? super T> c) {
```

Bonus: si utilizzi il metodo `comparatorBy()` per ottenere l'oggetto `Comparator` desiderato.

(c) 4 punti (0.5 punti per domanda) Per ciascuna dei seguenti binding Java si indichi con una crocetta l'esito della compilazione[2]:

`Comparator<Cube> cmpCube = Solid.comparatorBy(Cube::perimeter)`	◯ si	◯ no
`Comparator<Solid> cmpSolid = Solid.comparatorBy(Solid::area)`	◯ si	◯ no
`Comparator<Sphere> cmpSphere = Solid.comparatorBy(Sphere::perimeter)`	◯ si	◯ no
`Comparator<Solid> cmpSolid2 = Solid.comparatorBy(Cube::area)`	◯ si	◯ no
`Comparator<Polyhedron> cmpPoly = Solid.comparatorBy(Polyhedron::volume)`	◯ si	◯ no
`Comparator<Sphere> cmpSphere2 = Solid.comparatorBy(Solid::area)`	◯ si	◯ no
`Comparator<Polyhedron> cmpPoly2 = Solid.comparatorBy(Solid::volume)`	◯ si	◯ no
`Comparator<Cube> cmpCube2 = Solid.comparatorBy(Polyhedron::perimeter)`	◯ si	◯ no

(d) 4 punti (0.5 punti per domanda) Per ciascuna delle seguenti espressioni Java si indichi con una crocetta l'esito della compilazione. Si assumano nello scope le variabili di tipo `Comparator` di cui all'esercizio precedente.

`Collections.sort(solids, cmpCube2)`	◯ si	◯ no
`Collections.sort(cubes, cmpSolid)`	◯ si	◯ no
`Collections.sort(spheres, cmpSphere2)`	◯ si	◯ no
`Collections.sort(solids, cmpPoly2)`	◯ si	◯ no
`Collections.sort(cubes, cmpSolid2)`	◯ si	◯ no
`Collections.sort(cubes, cmpPoly)`	◯ si	◯ no
`Collections.sort(spheres, cmpPoly2)`	◯ si	◯ no
`Collections.sort(polys, cmpSolid)`	◯ si	◯ no

(e) 4 punti Si implementi uno snippet di codice Java che, per ogni poliedro della lista `polys`, lo posiziona in un punto dello spazio a piacere e poi ne stampa in standard output tutti i punti.

Question:	1	2	3	4	5	Total
Points:	2	2	6	6	17	33
Bonus Points:	0	2	1	0	0	3
Score:						

[2] Ricordiamo la sintassi dei *method reference*: sia T un tipo e m il nome di un metodo non statico della classe T, allora l'espressione $T::m$ computa una funzione avente tanti parametri quanti sono i parametri nella firma del metodo m, più un parametro extra di tipo T che rappresenta l'oggetto sul quale invocare il metodo. Tale parametro extra di tipo T compare come primo parametro; il tipo di ritorno della funzione è lo stesso tipo di ritorno di m. Se m si riferisce ad un metodo non statico senza parametri, la funzione risultante è unaria ed ha solamente il parametro extra di tipo T. Ad esempio, si assuma un oggetto `c` di tipo `Cube`, allora il binding `Function<Cube, Double> f = Cube::area` è valido e l'espressione `f.apply(c)` ritorna l'area di `c` esattamente come l'invocazione diretta `c.area()`.