Programmazione ad Oggetti
Mod. 2

3/6/2022

Studente _____ Matricola _____

1. Definiamo in Java 8+ un sistema di classi ed interfacce che rappresentano figure geometriche piane e solide. Le figure geometriche rappresentate non sono posizionate nel piano cartesiano o nello spazio, sono pertanto prive di coordinate. Per semplicità esse contengono solamente le informazioni sulla lunghezza dei lati o delle facce di cui sono costituite.

   Prima di cominciare, realizziamo una piccola libreria interna che consiste in alcuni metodi statici generici altamente riusabili. Sia data la funzione di ordine superiore `fold`[1] implementata tramite un metodo statico pubblico:

   ```
   public static <T, State> State fold(Iterable<T> i, final State st0, BiFunction<State, T, State> f) {
       State st = st0;
       for (final T e : i)
           st = f.apply(st, e);
       return st;
   }
   ```

   (a) [1 punti] Si implementi tramite **una sola** invocazione di `fold()` la funzione di ordine superiore `sumBy` avente la seguente firma:

   ```
   public static <T> double sumBy(Iterable<T> i, Function<T, Double> f);
   ```

   Essa calcola la sommatoria di tutti gli elementi di `i` trasformandoli in `double` tramite `f`. La utilizzeremo ad esempio per calcolare il perimetro di un poligono sommando la lunghezza di tutti i suoi lati, oppure l'area laterale totale di un solido sommando l'area di tutte le superfici piane di cui è costituito.

   (b) [1 punti] Avremo bisogno di ordinare le nostre figure geometriche sulla base di diversi criteri, ad esempio l'area o il volume. Si implementi il metodo statico `compareBy()` avente la seguente firma:

   ```
   public static <T> int compareBy(T s1, T s2, Function<T, Double> f);
   ```

   Esso confronta `s1` ed `s2` conventerdoli prima in `double` tramite `f`, riducendo pertanto il confronto al confronto tra due numeri `double`.

   (c) [1 punti (bonus)] Quale forma di polimorfismo forniscono i *generics* definiti sulla firma di un metodo come ad esempio quelli del metodo `fold()` di cui sopra?

   ◯ Polimorfismo subtype ◯ Polimorfismo parametrico ◯ Polimorfismo parametrico first-class
   ◯ Polimorfismo ad-hoc

2. Definiamo ora i tipi essenziali per rappresentare figure geometriche piane e solide.

   (a) [1 punti] La classe `Edge` rappresenta grandezze 1-dimensionali come lati di poligoni, spigoli di poliedri, segmenti e circonferenze.

   ```
   public class Edge implements Comparable<Edge> {
       private final double len;
       public Edge(double len) { this.len = len; }
       public double length() { return len; }
       @Override
   ```

[1] La funzione `fold` è nota nel mondo dei linguaggi imperativi mainstream con il nome di `reduce` ed è simile al metodo `Stream.reduce()` incluso nel JDK dalla versione 8 in poi.

```
    public int compareTo(Edge s) { /* DA IMPLEMENTARE */ }
    }
}
```

Si implementi il metodo `compareTo()` tramite **una sola** invocazione della `compareBy()` definita sopra.

(b) [1 punti] L'interfaccia `Surface` rappresenta figure piane qualunque:

```
public interface Surface extends Comparable<Surface> {
    double area();
    double perimiter();
    @Override
    default int compareTo(Surface s) { /* DA IMPLEMENTARE */ }
}
```

Si dia una implementazione di default del metodo `compareTo()` tramite **una sola** invocazione della `compareBy()` definita sopra che esegua il confronto tra le aree.

(c) [1 punti] Il sotto-tipo `Polygon` rappresenta poligoni: l'interfaccia specializza `Surface` dando una implementazione di default al metodo `perimeter()` e permette anche l'iterazione dei lati di cui il poligono stesso è costituito.

```
public interface Polygon extends Surface, Iterable<Edge> {
    @Override
    default double perimiter() { /* DA IMPLEMENTARE */ }
}
```

Si dia una implementazione di default del metodo `perimeter()` tramite **una sola** invocazione della `sumBy()` definita sopra.

(d) [1 punti] L'interfaccia `Solid` rappresenta solidi qualunque:

```
public interface Solid extends Comparable<Solid> {
    double outerArea(); // area laterale totale
    double volume();
    @Override
    default int compareTo(Solid s) { /* DA IMPLEMENTARE */ }
}
```

Si implementi il metodo `compareTo()` tramite **una sola** invocazione della `compareBy()` definita sopra che esegua il confroto tra i volumi.

(e) [1 punti] `Polyhedron` è sottotipo di `Solid` e rappresenta poliedri. L'interfaccia è parametrica rispetto al sottotipo di `Polygon` che descrive le facce di cui il poliedro è costituito. Ad esempio, le facce di un cubo sono quadrati: una classe `Cube` implementerebbe pertanto l'interfaccia `Polyhedron` avente `Square` come *type argument*, assumendo che esista `Square` $\preceq$ `Polygon`[2].

Un `Polyhedron` permette l'iterazione delle facce poligonali di cui esso è costituito; fornisce inoltre una implementazione di default del metodo `outerArea()` che calcola semplicemente la sommatoria delle aree delle sue facce.

```
public interface Polyhedron<P extends Polygon> extends Solid, Iterable<P> {
    @Override
    default double outerArea() { /* DA IMPLEMENTARE */ }
}
```

Si implementi il metodo `outerArea()` tramite **una sola** invocazione della `sumBy()` definita sopra.

3. Si proceda ora alla definizione di una gerarchia di classi che rappresentano figure geometriche specifiche implementando le interfacce fin qui introdotte.

(a) [2 punti] Si implementi una classe che rappresenta *sfere* immutabili avente nome `Sphere` e che implementa l'interfaccia `Solid`. Il costruttore di `Sphere` deve prendere come parametro solamente un `double`: il raggio della sfera[3].

[2] Siano τ e σ due tipi, allora la relazione di subtyping $\tau \preceq \sigma$ indica che τ è sottotipo di σ.
[3] Si ricordi che una sfera di raggio r ha volume $\frac{4}{3}\pi r^3$ e area laterale totale $4\pi r^2$.

(b) [2 punti] Si implementi una classe che rappresenta *cilindri* immutabili avente nome `Cilinder` e che implementa l'interfaccia `Solid`. Il costruttore di `Cilinder` deve prendere come parametri due `double`: il raggio della base e l'altezza del cilindro[4].

(c) [2 punti] Si implementi una classe che rappresenta *rettangoli* immutabili avente nome `Rectangle` e che implementa l'interfaccia `Polygon`. Il costruttore di `Rectangle` deve prendere come parametri due `double`: base e altezza.

(d) [2 punti] Si implementi una classe che rappresenta *quadrati* immutabili avente nome `Square` e che estende `Rectangle`. Il costruttore di `Square` deve prendere un solo parametro di tipo double: il lato.

4. Si prenda in considerazione la seguente classe che rappresenta parallelepipedi immutabili. I parallelepipedi sono poliedri aventi facce rettangolari.

```
public class Parallelepiped implements Polyhedron<Rectangle> {
    protected final double width, height, depth;
    public Parallelepiped(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
    @Override
    public double volume() { /* DA IMPLEMENTARE */ }
    @Override
    public Iterator<Rectangle> iterator() {
        Rectangle r1 = new Rectangle(width, height),
                  r2 = new Rectangle(width, depth),
                  r3 = new Rectangle(height, depth);
        return List.of(r1, r2, r3, r1, r2, r3).iterator();
    }
}
```

(a) [1 punti] Si implementi il metodo `volume()`.

(b) [1 punti] Si definisca la classe `Cube` come sottoclasse di `Parallelepiped`. Il costruttore di `Cube` deve prendere solamente un parametro di tipo `double`: la lunghezza dello spigolo.

(c) [1 punti (bonus)] E' possibile co-variare il tipo di ritorno del metodo `iterator()` di `Cube` in modo che ritorni un `Iterator<Square>`? Si motivi la risposta.

(d) [1 punti] Assumendo l'implementazione di `Cube` di cui al punto (b), si prenda in considerazione il seguente snippet:

```
1  int facet_cnt = 1;
2  for (Square sq : new Cube(10.)) {
3      int side_cnt = 1;
4      for (Edge e : sq) {
5          System.out.printf("side #%d/%d = %f\n", side_cnt++, facet_cnt, e.length());
6      }
7      ++facet_cnt;
8  }
```

Tale codice compila? Si motivi la risposta.

5. [6 punti] Si prenda in considerazione il seguente snippet di codice che crea una lista eterogena di poliedri le cui facce sono almeno rettangoli.

```
1  Cube c1 = new Cube(1.), c2 = new Cube(2.));
2  Parallelepiped p1 = new Parallelepiped(1., 2., 3.), p2 = new Parallelepiped(2., 3., 4.);
3  List<Polyhedron<? extends Rectangle>> polys = List.of(c1, c2, p1, p2);
```

[4] Si ricordi che un cilindro è l'estrusione di un cerchio nello spazio. L'area di un cerchio di raggio r è pari a πr^2; la sua circonferenza $2\pi r$.

Seguono ora alcune chiamate ai metodi `Collections.sort()` del JDK. Gli oggetti nella lista `polys` sono sempre gli stessi 4 (i cubi `c1` e `c2` ed i parallelepipedi `p1` e `p2`), ma ogni invocazione di `sort()` li ordina in maniera diversa a seconda del criterio di ordinamento. Per ciascuna invocazione si indichi quale oggetto viene messo in testa alla lista, ovvero quale oggetto computerebbe l'espressione `polys.get(0)`, oppure se non compila.

```
Collections.sort(polys)
```
◯ non compila ◯ c1 ◯ c2 ◯ p1 ◯ p2

```
Collections.sort(polys, (x, y) -> compareBy(x, y, Polyhedron::outerArea))
```
◯ non compila ◯ c1 ◯ c2 ◯ p1 ◯ p2

```
Collections.sort(polys, (x, y) -> compareBy(x, y, (p) -> p.outerArea()))
```
◯ non compila ◯ c1 ◯ c2 ◯ p1 ◯ p2

```
Collections.sort(polys, (x, y) -> compareBy(x, y, (r) -> r.perimiter()))
```
◯ non compila ◯ c1 ◯ c2 ◯ p1 ◯ p2

```
Collections.sort(polys, (x, y) -> compareBy(x, y, new Function<>() {
    public Double apply(Polyhedron<? extends Rectangle> r) {
        return r.volume();
    }
}))
```
◯ non compila ◯ c1 ◯ c2 ◯ p1 ◯ p2

```
Collections.sort(polys, (x, y) -> Double.compare(sumBy(x, Square::perimiter),
                                                 sumBy(y, Rectangle::perimiter)))
```
◯ non compila ◯ c1 ◯ c2 ◯ p1 ◯ p2

6. 7 punti Si scriva in C++[5] una classe generica `matrix` che rappresenta matrici bidimensionali di valori di tipo `T`, dove `T` è un tipo templatizzato. Tale classe deve comportarsi come un *valore*, secondo lo stile dei *container* di STL, pertanto deve supportare il costruttore per copia, l'operatore di assegnamento ed il costruttore di default. Si aggiungano poi metodi e operatori a piacere, tra cui **almeno** i seguenti:

   - costruttore con 2 parametri + 1 opzionale: numero di righe, numero di colonne e valore iniziale di tipo `T`;
   - distruttore (se necessario);
   - member type `iterator`, `const_iterator` e `value_type`;
   - accesso tramite riga e colonna implementando 2 overload (const e non-const) di `operator()`;
   - 2 overload (const e non-const) dei metodi `begin()` ed `end()` per poter iterare tutta la matrice dall'angolo superiore sinistro all'angolo inferiore destro come se fosse piatta.

   L'implementazione può utilizzare STL liberamente, se lo si desidera.

   Si prenda a riferimento il seguente snippet per la specifica dei requisiti del tipo `matrix`:

```
matrix<double> m1;          // non inizializzata
matrix<double> m2(10, 20);  // 10*20 inizializzata col default constructor di double
matrix<double> m3(m2);      // costruita per copia
m1 = m2;                    // assegnamento
m3(3, 1) = 11.23;           // operatore di accesso come left-value
for (typename matrix<double>::iterator it = m1.begin(); it != m1.end(); ++it) {
    typename matrix<double>::value_type& x = *it;  // de-reference non-const
    x = m2(0, 2);  // operatore di accesso come right-value
}
matrix<string> ms(5, 4, "ciao"); // 5*4 inizializzata col la stringa passata come terzo argomento
for (typename matrix<string>::const_iterator it = ms.begin(); it != ms.end(); ++it)
    cout << *it;   // de-reference const
```

[5] Si utilizzi la revisione del linguaggio C++ che si preferisce, motivando tuttavia qualunque scelta superiore al C++ vanilla.