Esercizio 1

Considerate la seguente gerarchia di classi:

```
class A {
public void print(String s) { System.out.println(s); }
public void m(int i) { print("A.m(int)"); }
public void m(boolean b) { print("A.m(boolean)"); }
}
public A k() { return this; }
}
class B extends A {
public void m(float f) { print("B.m(float)"); }
public void m(boolean b) { print("B.m(boolean)"); }
}
class C extends A {
public void m(int i) { print("C.m(int)"); }
public void m(double d) { print("C.m(double)"); }
}
```

e assumete le seguenti dichiarazioni di variabile.

```
 A va = new A();  A vab = new B();
 B vb = new B();  A vac = new C();
 C vc = new C();
```

Indicare l'output prodotto dalle seguenti espressioni. Se il comando produce più di una linea di output, utilizzate il carattere '/' per indicare le diverse linee. Se il comando causa errore, indicate il tipo di errore (compilazione o run-time).

1. `vab.m(1);`
2. `vab.m(1.2);`
3. `vac.m(1);`
4. `vb.m(1.2);`
5. `((B)vab).m(1.2);`
6. `((B)(vab.k())).m(1);`
7. `((B)(va.k())).m(1);`
8. `((C)(vac.k())).m(false);`
9. `((B)vb.k()).m(true);`
10. `vac.k().m(1.2);`

Soluzione Esercizio 1

1. `vab.m(1);` `A.m(int)`
2. `vab.m(1.2);` compiler error
3. `vac.m(1);` `C.m(int)`
4. `vb.m(1.2);` `B.m(float)`
5. `((B)vab).m(1.2);` `B.m(float)`
6. `((B)(vab.k())).m(1);` `A.m(int)`
7. `((B)(va.k())).m(1);` runtime error
8. `((C)(vac.k())).m(false);` `A.m(boolean)`
9. `((B)vb.k()).m(true);` `B.m(boolean)`
10. `vac.k().m(1.2);` compiler error

Esercizio 2

Implementare due rappresentazioni diverse del concetto di un punto sul piano, rispettivamente in coordinate cartesiane e polari. Ricordiamo brevemente che nel sistema di coordinate polari un punto è descritto da due valori: il raggio r che rappresenta la distanza dall'origine, e l'angolo θ che rappresenta l'angolo in senso antiorario formato dal raggio a partire dall'asse di ascissa. Assumiamo che tutti gli angoli siano espressi in radianti, utilizzando la convenzione per cui $2\pi\,\text{rad} = 360°$. Le conversioni tra rappresentazione cartesiana e polare sono definite dalle seguenti equazioni:

$$\begin{aligned} x &= r\cos\theta & r &= \sqrt{x^2 + y^2} \\ y &= r\sin\theta & \theta &= \text{atan2}(x, y) \end{aligned}$$

Sia data la seguente interfaccia:

```
interface Point
{
  /**
  * restituisce la coordinata x di questo punto
  */
  double xCoordinate();

  /**
  * restituisce la coordinata y diquesto punto
  */
  double yCoordinate();

  /**
  * restituisce la distanza dall'origine diquesto punto
  */
  double radius();

  /**
  * restituisce la misura  in radianti dell'angolo
  * (in senso antiorario) dall'asse x
  */
  double angle();

  /**
  * restituisce un nuovo punto ottenuto da una rotazione di
  * d gradi (in radianti) in senso antiorario dall'origine
  */
  public Point rotate(double d);

  /**
  * restituisce un nuovo punto ottenuto da una traslazione di
  * (dx,dy) the punto attuale. Cio\`e le coordinate x ed y sono
  * uguali alle coordinate x e y del punto attuale a cui vengono
  * sommati i valori dx e dy rispettivametne
  */
  public Point translate(double dx, double dy);
}
```

1. Definite due classi `CartesianPoint` e `PolarPoint`. per realizzare le due rappresentazioni. Entrambe le classi devono implementare `Pont`. La classe `CartesianPoint`, ha un costruttore con la seguente struttura:
   `CartesianPoint(double xCoord, double yCoord)`
   che crea un nuovo CartesianPoint con le coordinate definite dai due parametri. Nel caso il punto sia alle coordinate $(0, 0)$, entrambi i metodi `radius()` e `angle()` restitu- iscono 0.0. La classi `PolarPoint`, a sua volta, ha un costruttore con la seguente struttura:
   `PolarPoint(double radius, double angle)`
   che crea un nuovo `PolarPoint` con raggio `radius` e angolo `angle` espresso in radianti. Se il raggio è negativo, inverte il valore e modifica l'angolo (ruotando di π in senso antiorario); se l'angolo è negativo, ruota di un numero di gradi (in radianti) sufficienti a renderlo posi tivo). In tutti i casi, la rappresentazione deve garantire che il raggio sia positivo, e l'angolo compreso nell'intervallo $[0 \ldots 2\pi]$;

2. Create una classe applicazione per testare la vostra implementazione. La classe definisce:

- due metodi: `distance()` che calcola la distanza tra due punti, e `sameQuadrant()` che determina se due punti stanno nello stesso quadrante;
- un metodo main che testa le implemetazioni utilizzando i due metodi della classe.

Soluzione Esercizio 2

```
class CartesianPoint implements Point
{
  private double x;
  private double y;

  public CartesianPoint(double xCoord, double yCoord) { x=xCoord; y=yCoord; }

  public double xCoordinate() { return x; }
  public double yCoordinate() { return y; }
  public double radius() { return Math.sqrt(x*x+y*y); }
  public double angle()
  {
    if (radius()==0.0) return 0.0;
    else return Math.atan2(x,y);
  }
  public Point rotate(double d) { return new PolarPoint(radius(), angle()+d); }
  public Point translate(double dx, double dy)
  { return new CartesianPoint(xCoordinate()+dx,yCoordinate()+dy); }
}

class PolarPoint implements Point
{
  private double r;
  private double theta;

  public PolarPoint(double radius, double angle)
  {
    r=radius;
    theta=angle;
    if (r<0)
    {
      r=-r;
      theta = theta + Math.PI;
    }

    int turns = (int)(theta/(Math.PI*2));
    if (theta<0.0) turns = turns -1;

    theta = theta - turns*Math.PI*2;
  }

  public double xCoordinate() { return r*Math.cos(theta); }
  public double yCoordinate() { return r*Math.sin(theta); }
  public double radius() { return r; }
  public double angle() { return theta; }
  public Point rotate(double d) { return new PolarPoint(radius(), angle()+d); }
  public Point translate(double dx, double dy)
  { return new CartesianPoint(xCoordinate()+dx,yCoordinate()+dy); }
}


class PointApp
{
  public static double distance(Point a, Point b)
  {
```

```
    Point diff=a.translate(-b.xCoordinate(),-b.yCoordinate());
    return diff.radius();
  }

  public static boolean sameQuadrant(Point a, Point b)
  {
    // due punti appartengono allo stesso quadrante se tutte le coordinate hanno le stesso segno (lo 0 e‘
    // testo che le coordinate abbiano lo stesso segno controllando che il prodotto sia >= 0
    return (a.xCoordinate()*b.xCoordinate() >= 0.0) && (a.yCoordinate()*b.yCoordinate() >= 0.0);
  }

  public static void main(String[] args)
  {
    Point a = new CartesianPoint(1, 2);
    Point b = new PolarPoint(1, 0.1);

    double dist=distance(a,b);
    boolean q=sameQuadrant(a,b);

    System.out.println("distance: " + dist);
    System.out.println("same quadrant: "  + q);
  }
}
```

Esercizio 3

Sia data la seguente interfaccia

```
interface Predicate<T>
{ boolean evaluate(T e); }
```

Implementare il metodo `filter` che prenda in input un `Predicate<T>` e un `Iterator` su di un sottotipo di `T` e restituisca un iteratore che iteri sui soli elementi dell'iteratore in ingresso che soddisfino al predicato (`evaluate` su quegli elementi dia `true`).
Ricordiamo qui di seguito la specifica dellinterfaccia `Iterator<T>`.

```
interface Iterator<E>
{
  /**
  * Restituisce ture se l'iteratore ha ancora elementi disponibili
  */
  boolean hasNext();

  /**
  * Restituisce il sucessico elemento nell'iterazione.
  * Lancia NoSuchElementException se non ci sono elementi disponibili
  */
  E next();

  /**
  * Rimuove dalla collezione l'ultimo elemento ritornato dall'iteartore (operazione opzionale).
  * Se l'operazione non e` supportata lancia UnsupportedOperationException.
  * Se l'operazione e` supportata ma next() non e` ancora stato chiamato, lancia IllegalStateException.
  */
  void remove();
}
```

Nota: **non** è obbligatorio implementare un nuovo iteratore.

Soluzione Esercizio 3

La soluzione più rapida, senza implementare un nuovo iteratore è:

```
  public static <T, S extends T> Iterator<S> filter(Predicate<T> pred, Iterator<S> iter) {
    ArrayList<S> filtered=new ArrayList<S>();
    while(iter.hasNext()) {
      S current=iter.next();
      if (pred.evaluate(current)) filtered.add(current);
      }
    return filtered.iterator();
    }
```

Nel caso si voglia implementare un nuovo iteratore:

```
class filteredIterator<T,S extends T> implements Iterator<S> {
  private Iterator<S> base_iterator;
  private Predicate<T> predicate;
  S current;

  public filteredIterator(Predicate<T> pred, Iterator<S> iter) {
    predicate=pred;
    base_iterator=iter;
    getCurrent();
    }

    private void getCurrent() {
    current=null;
    while (current==null && base_iterator.hasNext()) {
      S val=base_iterator.next();
```

```
      if (predicate.evaluate(val)) current=val;
      }
    }

  public boolean hasNext() { return current!=null; }

  public S next() {
    S result=current;
    getCurrent();
    return result;
    }

  public void remove() { throw new UnsupportedOperationException(); }
  }
```

e il metodo `filter` diventa semplicemente:

```
  public static <T, S extends T> Iterator<S> filter(Predicate<T> pred, Iterator<S> iter) {
    return new filteredIterator<T,S>(pred,iter);
    }
```

Esercizio 4

Si scriva una gerarchia di classi che rappresenti una composizione arbitraria di resistenze in un circuito elettrico in termini di resistenze atomiche e composizioni in serie ed in parallelo di resistenze. La gerarchia dovrà offire un metodo polimorfico `double calcola()` che restituisce il valore della resistenza effettiva del circuito. Si ricordi che la resistenza effettiva R_e di due composizioni di resistenze messe in serie tra di loro e aventi resistenze effettive R_1 ed R_2, è $R_e = R_1 + R_2$. Per due composizioni di resistenze messe in paralleo vale $\frac{1}{R_e} = \frac{1}{R_1} + \frac{1}{R_2}$.

Soluzione Esercizio 4

```
interface Resistenza {
  double calcola();
}

class ResistenzaSingola implements Resistenza {
  private double valore;
  public ResistenzaSingola( double v ) { valore=v; }
  public double calcola() { return valore; }
}

class ResistenzeInSerie implements Resistenza {
  private Resistenza prima, dopo;
  public ResistenzeInSerie( Resistenza p, Resistenza d ) { prima=p; dopo=d; }
  public double calcola() { return prima.calcola() + dopo.calcola(); }
}

class ResistenzeInParallelo implements Resistenza {
  private Resistenza destra, sinistra;
  public ResistenzeInSerie( Resistenza dx, Resistenza sx ) { destra=dx; sinistra=sx; }
  public double calcola() { return 1.0/(1.0/destra.calcola() + 1.0/sinistra.calcola()); }
}
```

Esercizio 5

Sia data la classica gerarchia di forme:

```
interface Shape
{
  void draw(Raster o);
}

class Square implements Shape {...}
class Circle implements Shape {...}
class Triangle implements Shape {...}
class Rectangle implements Shape {...}
class Ellipse implements Shape {...}
...
```

dove il metodo `draw` disegna l'interno della forma (forma "piena").
L'interfaccia `Raster` fornisce un modo semplificato per disegnare linee orizzontali

```
interface Raster
{
// disegna una linea orizzontale sulla riga "row" a partire dalla colonna "startColumn"
// fino alla colonna "endColumn"
void drawSegment(int row, int startColumn, int endColumn);
}
```

Date queste classi base che si considerano non modificabili, siete stati commissionati per scrivere del codice che disegni le forme geometriche con un tratteggio a 45° (dall'alto a destra, verso il basso a sinistra) con pieni e vuoti della stessa dimensione (vedere Figura 1).

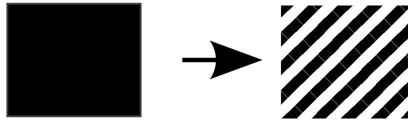

Figura 1: Rettangolo pieno e rettangolo tratteggiato.

Si commenti la soluzione adottata, con particolare attenzione alla scalabilità, alla duplicazione e alla possibilità di riutilizzo dell'approccio a nuove estensioni.