Esercizio 5

Sia data la seguente classe che rappresenta un conto bancario

```
class Conto
{
  private double saldo;
  public Conto(double saldo) throws IllegalArgumentException
  {
    if (saldo < 0)
      throw new IllegalArgumentException("saldo negativo");
    this.saldo = saldo;
  }
  public void preleva (double importo) throws IllegalArgumentException
  {
    if (saldo <= 0)
      throw new IllegalArgumentException("importo non positivo");
    if (saldo < importo)
      throw new IllegalArgumentException("importo > saldo");
    saldo -= importo;
  }
  public void deposita(double importo) throws IllegalArgumentException
  {
    if (importo <= 0)
      throw new IllegalArgumentException("importo non positivo");
    saldo += importo;
  }
}
```

Vogliamo definire una sottoclasse `ContoPlus` che permette di tracciare le operazioni di prelievo e deposito su ciascun conto, e fornisce un metodo per ottenere l'estratto conto con il dettaglio di tali operazioni. Definite:

- una classe `Operazione`: la classe è non istanziabile e contiene un campo (immutabile) pari al valore assoluto in euro dell'operazione e la data dell'operazione (usate la classe `java.util.Date` che ha un costruttore `Date()`, senza parametri, che restituisce la data corrente). La classe deve definire il costruttore necessario a inizializzare i suoi campi, ed un metodo astratto che restituisce l'importo positivo/negativo dell'operazione (positivo se i soldi vanno aggiunti al conto, negativo se i soldi vanno sottratti);
- due sottoclassi concrete di `Operazione`: `Prelievo` e `Versamento` che descrivono le corrispondenti operazioni. Definite i costruttori di queste classi (riusando quello della superclasse) ed implementate i metodi astratti di `Operazione`;
- la classe `ContoPlus`, sottoclasse di `Conto`, che definisce un metodo pubblico per calcolare l'estratto conto come specificato qui di seguito:

  ```
  /**
  * restituisce la lista delle operazioni del conto effettuate
  * nel periodo specificato. Se la data di inizio e‘ maggiore
  * di quella di fine, il metodo lancia l’eccezione
  */
  public List<Operazione> estrattoConto(Date inizio, Date fine)
  throws IllegalDatesException
  ```

  La classe `ContoPlus` deve inoltre definire un construttore e ridefinire i metodi della superclasse (riuti- lizzando quelli della stessa superclasse) per permettere di tener traccia delle operazioni.

  Nota: la classe `Date` implementa l'interfaccia `Comparable<Date>`, ha quindi un metodo publico `int compareTo(Date e)` che restituisce un intero minore, uguale o maggiore di zero se, rispettivamente, `this` è minore, uguale o maggiode di `e`.

Soluzione

```
abstract class Operazione
{
  private double valore;
  private Date date;

  protected double get_valore() { return valore ;}

  public Operazione(Date d, double v) { date=d; valore=v; }

  public Date data() { return date; }
  public abstract double importo();
}

class Prelievo extends Operazione
{
  public Prelievo(Date d, double v) { super(d,v); }
  public double importo() { return -get_valore(); }
}

class Deposito extends Operazione
{
  public Deposito(Date d, double v) { super(d,v); }
  public double importo() { return get_valore(); }
}

class ContoPlus extends Conto
{
  private ArrayList<Operazione> operazioni = new ArrayList<Operazione>();

  public ContoPlus(double saldo) throws IllegalArgumentException
  {
    super(saldo);
  }

  public void preleva (double importo) throws IllegalArgumentException
  {
    super.preleva(importo);
    operazioni.add(new Prelievo(new Date(), importo));
  }

  public void deposita(double importo) throws IllegalArgumentException
  {
    super.deposita(importo);
    operazioni.add(new Deposito(new Date(), importo));
  }

  public List<Operazione> estrattoConto(Date inizio, Date fine) throws IllegalDatesException
  {
    if ( inizio.compareTo(fine) > 0 ) throw new IllegalDatesException();

    ArrayList<Operazione> result = new ArrayList<Operazione>();

    for (Operazione e : operazioni)
    {
      if ( inizio.compareTo(e.data())<=0 && fine.compareTo(e.data())>=0 )
result.add(e);
    }
    return result;
  }
}
```

Esercizio 6

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:

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

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

```
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 7

Scrivere una classe applicazione `ThresholdIteratorApp` che fornisca il seguente metodo

```
/**
* Restituisce un iteratore che a sua volta rstituisce
* in sequenza tutti gli elementi della lista l che
* hanno un valore non inferiore (>=) a soglia.
* Non crea una nuova lista, e non modifica la lista l
*/
public static Iterator<Integer> iterator(List<Integer> l, int soglia)
```

Il metodo non deve creare un *container* ausiliario, ma deve creare una classe che implementi `Iterator<Integer>` e che filtri i valori di `l`
`ThresholdIteratorApp` deve fornire anche un metodo `main` che testi l'iteratore.
Di seguito viene data la specifica della interfaccia `Iterator`

```
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();
}
```

Non è necessario supportare la *remove*, ma l'iteratore restituiro de `iterator` deve essere un iteratre valido.

Soluzione

```
class ThresholdIteratorApp {
public static Iterator<Integer> iterator(List<Integer> l, int soglia)
{
  class ThresholdIterator implements Iterator<Integer>
  {
    private Iterator<Integer> baseIterator;
    private int soglia;
    private Integer nextInt;

    private void findNext()
    {
    nextInt=null;
    while(baseIterator.hasNext() && nextInt==null)
      {
Integer tmp=baseIterator.next();
if (tmp>=soglia) nextInt=tmp;
      }
    }

    public ThresholdIterator(Iterator<Integer> i, int soglia)
    {
      baseIterator=i;
```

```
      this.soglia=soglia;
      findNext();
    }

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

    public Integer next()
    {
      if (nextInt==null) throw new NoSuchElementException();

      Integer result=nextInt;
      findNext();
      return result;
    }

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

return new ThresholdIterator(l.iterator(), soglia);
}

  public static void main(String[] args)
  {
  ArrayList<Integer> l=new ArrayList<Integer>();

  l.add(1);
  l.add(2);
  l.add(3);

  Iterator<Integer> i=iterator(l,2);
  while(i.hasNext())
    {
    System.out.println(i.next().toString());
    }
  }
}
```