Esercitazione 3

Gerarchie di classi

@andreamocci, @mariosangiorgio

1. Persone e studenti

Implementare le classi per rappresentare delle persone e degli studenti.

Vogliamo tenere traccia del nome, cognome e della data di nascita di una persona.

Gli studenti sono delle persone cui è associata una lista di esami sostenuti.

```
private final String firstName, lastName;
private final Date birthday;

public Person(String firstName,String lastName,Date birthday){
   ... /* <- altri check su validita' dei parametri ... */
   /* Altro controllo: il compleanno non puo' essere nel futuro */
   if (birthday.after(new Date() /* giorno e ora corrente */ ))
     throw new IllegalArgumentException(" ... ");
   this.firstName = firstName;
   this.lastName = lastName;
   /*
    * Date e' un tipo di dato mutabile. Per evitare che dall'esterno
    * qualcuno possa modificare il compleanno della persona dobbiamo
    * crearne una copia.
    */
   this.birthday = (Date) birthday.clone();
}

public Date getBirthday() {
   // Anche qui non vogliamo esporre un riferimento allo stato
   // interno, ma solamente una sua copia
   return (Date) birthday.clone();
}
```

`Clonable`

`clone()` è un metodo (`protected`) definito in `Object`, ma l'implementazione non fa altro che lanciare una eccezione (`CloneNotSupportedException`).

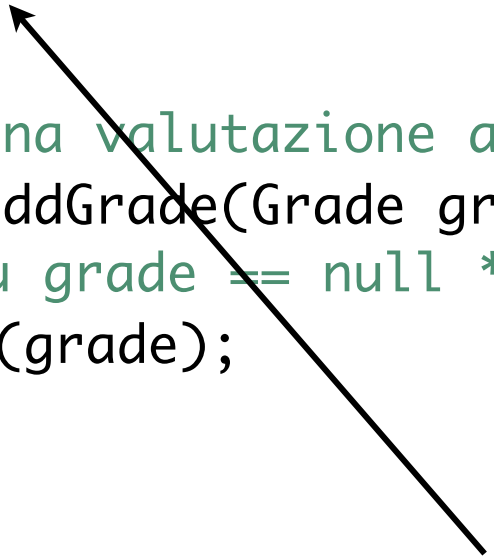
Se volete usarlo controllate che la classe implementi l'interfaccia `Clonable`.

Quando lo implementate ricordatevi di implementare `Clonable`. In alternativa, potete ottenere lo stesso risultato implementando un *copy-constructor*.

```java
public class Student extends Person {
   private final List<Grade> grades = new ArrayList<Grade>();

   public Student(String firstName, String lastName,
            Date birthday) {
      super(firstName, lastName, birthday);
   }

   // Aggiunge una valutazione alla carriera dello studente.
   public void addGrade(Grade grade) {
      /* check su grade == null */
      grades.add(grade);
   }
}
```



Richiama il costruttore della super-classe

`Collection` in Java

Nell'esempio di prima abbiamo creato una lista di voti di tipo: `List<Grade>`
Questo è un esempio dell'uso dei *generics*, classi il cui tipo è parametrico rispetto ad altri tipi.

`List` è una interfaccia, per creare un oggetto ci serve una classe concreta. Nell'esempio `ArrayList`

Perché è una buona idea usare il tipo più generale possibile (`List`) come tipo statico dell'attributo?

```java
// Controlla se lo studente ha abbastanza crediti per potersi laureare.
public boolean canGraduate() {
   return totalCredits() >= 180;
}

// Calcola la media pesata.
public double getWeightedGradeAverage() {
   double sumOfWeightedPoints = 0;
   for (Grade grade : grades) {
      sumOfWeightedPoints += grade.getCredits() * grade.getPoints();
   }
   return sumOfWeightedPoints / totalCredits();
}

// Calcola il numero di crediti sostenuti dallo studente.
private int totalCredits() {
   int totalCredits = 0;
   for (Grade grade : grades) {
      totalCredits += grade.getCredits();
   }
   return totalCredits;
}
```


Ciclo for generalizzato

Un altro uso di `super`

```
public class Person {
  @Override
  public String toString() {
    return "Hi, I'm " + firstName + " " + lastName;
  }
}

public class Student {
  @Override
  public String toString() {
    return super.toString() +
           " and I am a student";
  }
}
```

Tipi statici e dinamici

```
// Il costruttore di Date e' deprecato, lo usiamo
// solo per comodita' in questo esempio.
Person andrea = new Person("Andrea", "Mocci", new Date(1982, 6, 2));
Person studente = new Student("Dente", "Stu", new Date(1986, 11, 15));

System.out.println(andrea);
System.out.println(studente);
```

Qual è il tipo statico di ciascuna variabile?
Qual è il tipo dinamico?

Quale implementazione di `toString` viene invocata da `System.out.println`?

Cosa succederebbe se scrivessi l'istruzione `andrea.canGraduate()`

2. Forme geometriche

Implementare una gerarchia di classe che rappresenti delle forme geometriche nel piano, permettendo di calcolare la loro area e il loro perimetro.
Tra le figure implementare Cerchio e Quadrato. Nel caso del quadrato, deve essere possibile rappresentare un qualunque orientamento di questo rispetto all'asse delle ascisse.

```
public abstract class Shape {

   public abstract double getArea();
   public abstract double getPerimeter();

   }
```

L'implementazione di questi metodi dipende dalla specifica figura: devo dichiararli abstract. Tuttavia...

```
public abstract class Shape {

   public abstract double getArea();
   public abstract double getPerimeter();

   }
```

Come mai deve essere una classe astratta? C'è una giustificazione forte?

Square

```
public class Square extends Shape {

   private final Point initialPoint;
   // It represents the orientation of the square, in radiant degrees
   private final double orientation;
   private final double sideLength;

   public Square(Point initialPoint, double orientation, double sideLength) {
      if (initialPoint == null || sideLength < 0 ||
         orientation < 0 || orientation > 2 * Math.PI /* serve? */) {
         throw new IllegalArgumentException(...);
      }
      this.initialPoint = initialPoint;
      this.orientation = orientation;
      this.sideLength = sideLength;
   }
```

Square (osservatori)

```
@Override
public double getArea() {
   return sideLength * sideLength;
}

@Override
public double getPerimeter() {
   return 4 * sideLength;
}
```

Circle

```
public class Circle extends Shape {
   private final Point center;
   private final double radius;

   public Circle(Point center, double radius) {
      if (center == null || radius < 0) {
         throw new IllegalArgumentException();
      }
      this.center = center;
      this.radius = radius;
   }
```

Anche se cerchio e quadrato sono entrambi delle forme la loro rappresentazione è totalmente diversa

Circle osservatori

```
@Override
public double getArea() {
   return Math.PI * radius * radius;
}

@Override
public double getPerimeter() {
   return 2 * Math.PI * radius;
}
```

Rectangle

- Supponiamo di voler rappresentare un Rettangolo. Come e dove lo aggiungo alla gerarchia? Come cambia l'implementazione di Square?

Canvas

- Supponiamo di avere una classe TwoDimensionalCanvas che è in grado di disegnare segmenti nel piano (non ci interessa come...)

```
public class TwoDimensionalCanvas {
  ...
  public void drawLine(Point point, Point point2) {
  ...
  }
}
```

- Come possiamo fornire il supporto per il disegno delle forme?

```java
public abstract class Shape {
   public abstract double getArea();
   public abstract double getPerimeter();

   // espongo una rappresentazione come sequenza di punti
   public abstract List<Point> getSequenceOfPointsToDraw();

   /*
    * In questo caso, ogni figura puo' essere disegnata in
    * maniera indipendente dalla specifica classe
    * (posso mettere questo metodo in Shape)
    */
   public void draw(TwoDimensionalCanvas canvas) {
      List<Point> points = getSequenceOfPointsToDraw();
      for (int i = 0; i < points.size() - 1; i++) {
         canvas.drawLine(points.get(i), points.get(i + 1));
      }
   }
```

Circle - disegno

```
@Override
public List<Point> getSequenceOfPointsToDraw() {
    // Approssimiamo il cerchio con una figura regolare con 256 lati
    List<Point> points = new ArrayList<Point>();
    int numberOfSegments = 256;
    for (int i = 0; i < numberOfSegments; i++) {
        double newPointX = center.getX() +
                radius * Math.cos(2 * Math.PI * i / numberOfSegments);
        double newPointY = center.getY() +
                radius * Math.sin(2 * Math.PI * i / numberOfSegments);
        points.add(new Point(newPointX, newPointY));
    }
    return points;
}
```

Square - disegno

```
@Override
public List<Point> getSequenceOfPointsToDraw() {
   List<Point> points = new ArrayList<Point>();
   points.add(initialPoint);

   Point nonOrientedSecondPoint =
      new Point(initialPoint.getX(),
                initialPoint.getY() + sideLength);
   points.add(nonOrientedSecondPoint.rotate(initialPoint, orientation));

   Point nonOrientedThirdPoint =
      new Point(initialPoint.getX() + sideLength,
                initialPoint.getY() + sideLength);
   points.add(nonOrientedThirdPoint.rotate(initialPoint, orientation));

   Point nonOrientedFourthPoint =
      new Point(initialPoint.getX() + sideLength,
                initialPoint.getY());
   points.add(nonOrientedFourthPoint.rotate(initialPoint, orientation));

   return points;
}
```

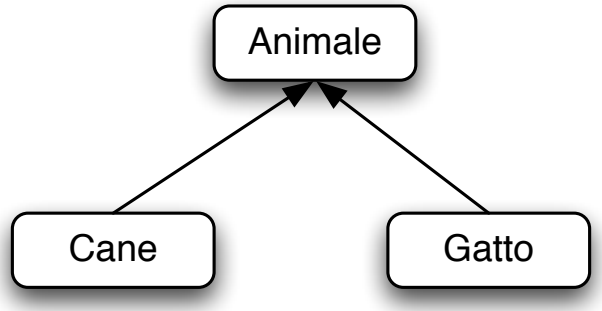
Problema

La figura deve esporre una rappresentazione compatibile con la classe esterna, e a volte, come per il cerchio, può essere una forzatura.

Il problema è derivato dai limiti di TwoDimensionalCanvas (disegna solo segmenti)

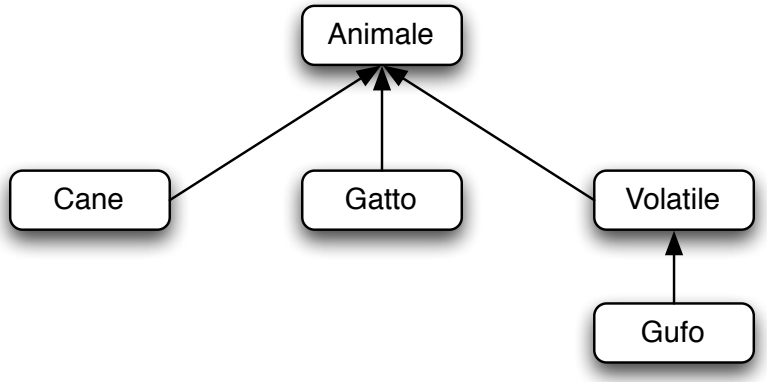
Ci sono meccanismi più eleganti per progettare situazioni simili a questa sfruttando il polimorfismo: li vedremo meglio quando faremo i design pattern.

3. Ereditarietà e interfacce



Come possiamo estendere questa gerarchia per supportare animali che possono volare?

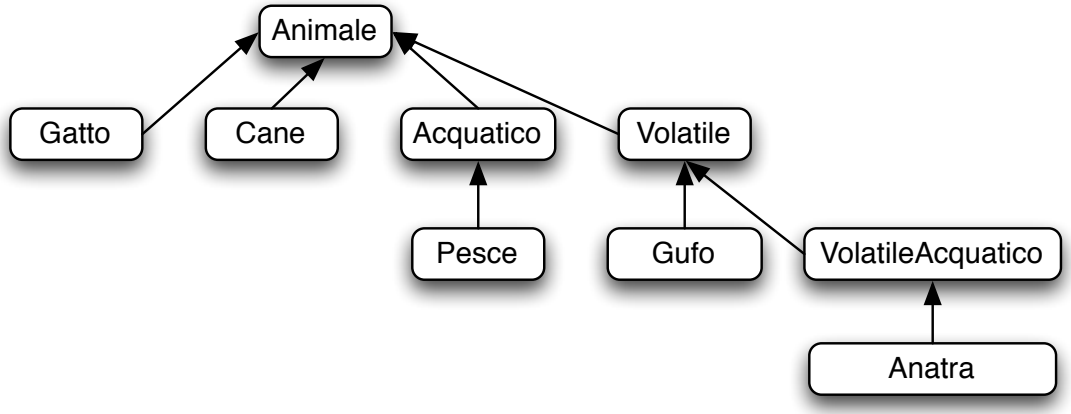
Possiamo introdurre
una nuova classe volatile



Può andare, ma cosa succederebbe se
volessimo rappresentare anche gli animali
che possono nuotare?

La gerarchia diventa molto complicata!

Una nuova classe di animali comporta modifiche rilevanti



Questa gerarchia ha anche un grave problema concettuale!

Non ci dice che *Pesce* e *Anatra* sono entrambi acquatici!

In altre parole, non c'è un tipo in comune abbastanza ristretto tra Pesce e Anatra che rappresenti il fatto che siano entrambi Acquatici

Soluzione: ereditarietà multipla

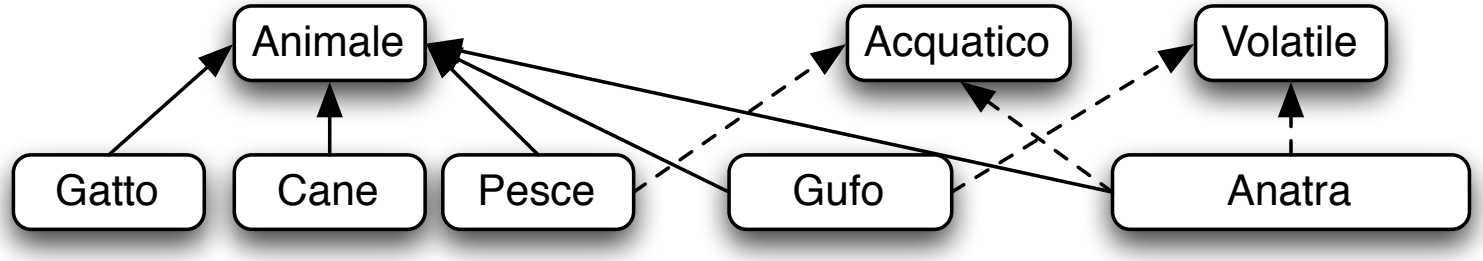
Concetti diversi sono separati e ciascuna classe può usare quelli che servono

In Java possiamo realizzarla usando le interfacce

Classi e Interfacce

Una classe (anche astratta) ci dice che un oggetto **è** qualcosa

Una interfaccia rappresenta un **comportamento** che la classe ha



VANTAGGI:

Gerarchia più semplice

Possiamo esprimere tutti i concetti che ci servono

Implementazione in Java

```
public class Anatra extends Animale implements Volatile, Acquatico {
    // Ha tutto ciò che aveva la classe Animale

    // Deve implementare i metodi definiti in Volatile e in Acquatico
    @Override
    public void vola(){ ... }

    @Override
    public void nuota(){ ... }
}
```

Un’anatra è un animale che può volare e nuotare

Interfacce come tipi

- Anche le interfacce possono essere usate come tipo di un riferimento
- Come tipo statico, un'interfaccia espone tutti i metodi che definisce
- L'implementazione utilizzata dipende dal tipo dinamico della classe (Binding dinamico)
- Per gli assegnamenti dobbiamo considerare le relazioni di tipo/sotto-tipo

Quali di queste operazioni sono valide?

```
Animale g = new Gatto();
Pesce p = new Pesce();
Anatra a = new Anatra();
Volatile g1 = g;
Volatile v = a;
Acquatico n = a;

g.vola();
g.nuota();
v.vola();
n.nuota();
```

Quali di queste operazioni sono valide?

```
Animale g = new Gatto();
Pesce p = new Pesce();
Anatra a = new Anatra();
Volatile g1 = g; // No
Volatile v = a;
Acquatico n = a;

g.vola();  //No, un animale generico non vola
g.nuota(); // e non nuota
v.vola();  //Sì, un voltatile può volare
n.nuota(); //Sì, un acquatico può nuotare
```