Classi e metodi `final`

- È possibile impedire la creazione di sottoclassi di una certa classe definendola **`final`**
- Esempio:

```
final class C {...}
class C1 extends C // errato
```

- Analogamente, è possibile impedire l'overriding di un metodo definendolo **`final`**
- Esempio:

```
class C { final void f() {...} }
class C1 extends C {
  void f() {...} // errato
}
```

Utile quando si vuole impedire di cambiare il comportamento di tutta o parte della classe

Attenzione…

```
public class Test {
  public static void main(String a[]) {
    Test c = new Derived();
    c.f1();
  }
  final void f1() { System.out.println("f1 in superclass");}
}
class Derived extends Test {
  public void f1() { System.out.println("f1 in subclass");}
}
```

Output (in compilazione…):

```
Test.java:13: error: f1() in Derived cannot override f1() in Test
 public void f1() { System.out.println("f1 in subclass"); }
             ^  overridden method is final
 1 error
```

Attenzione…

```
public class Test {
  public static void main(String a[]) {
    Test c = new Derived();
    c.f1();
  }
  private void f1() { System.out.println("f1 in superclass");}
}
class Derived extends Test {
  public void f1() { System.out.println("f1 in subclass");}
}
```

Output (a runtime):
f1 in superclass

- I metodi **`private`** non possono essere ridefiniti (come i **`final`**) ma sono completamente «invisibili» alle sottoclassi
- **`f1()`** in **`Derived`** è un nuovo metodo (no overriding)

Override di metodi `static`?

```
class C {
  static void m() {
    System.out.println("1");
  }
}
class D extends C {
  static void m() {
    System.out.println("2");
  }
}
```

```
(nel main)

C.m();          // 1
D.m();          // 2
C c = new C();
C cd = new D();
D d = new D();
c.m();          // 1
cd.m();         // 1
d.m();          // 2
```

i metodi `static` possono essere chiamati direttamente da una classe ...

... ma anche da un oggetto ...

... tuttavia, in tal caso il binding è statico e non dinamico!

Quale sarebbe l'output se `m()` non fosse un metodo `static`?

Una Pila polimorfa …

```
package strutture;
public abstract class VettoreDati {
   int size;
   int defaultGrowthSize;
   int marker;
   Object contenuto[];
   final int initialSize = 3;
   public VettoreDati() {
      size = initialSize;
      defaultGrowthSize = initialSize;
      marker = 0;
      contenuto = new Object[size];
   }
```

abilita lo static binding

Una Pila polimorfa …

```
final public void inserisci(Object k) {
   if(marker == size)
     cresci(defaultGrowthSize);
   contenuto[marker] = k;
   marker++;
}
abstract public Object estrai();
```

```
public Object estrai() {
   assert(marker>0):"Estrazione da Pila vuota";
   return contenuto[--marker];
}
```

In **Pila**

Una Pila polimorfa …

```
private void cresci(int dim){
   size += defaultGrowthSize;
   Object temp[] = new Object[size];
   for (int k=0; k<marker; k++)
    temp[k] = contenuto[k];
   contenuto = temp;
}
```

Una Pila polimorfa …

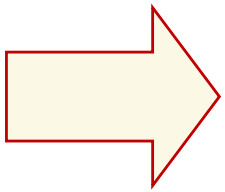
```
public static void main(String args[]) {
   int dim = 10;
   Pila s = new Pila(dim);
   for(int k=0; k<dim; k++){
      s.inserisci(k);
   }
   for (int k=0; k<3*dim; k++) {
      System.out.println(s.estrai());
   }
}
```

k è un **int**! Non posso sostituirlo a un **Object**…

Classi "wrapper"

tipi primitivi

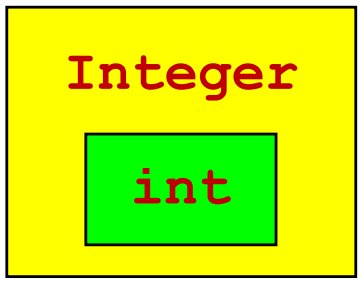
```
byte
short
int
long
float
double
char
boolean
```

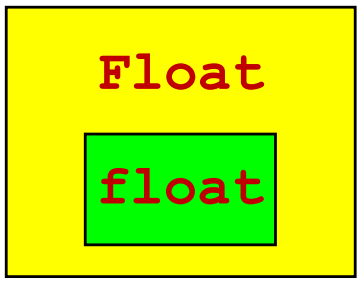


```
Byte
Short
Integer
Long
Float
Double
Char
Boolean
```

tipi riferimento classi «wrapper»

- Si usano per generare **oggetti** che contengono al loro interno un tipo di dato primitivo





Una Pila polimorfa …

```
public static void main(String args[]) {
   int dim = 10;
   Pila s = new Pila();
   for(int k=0; k<dim; k++){
      Integer o = new Integer(k);
      s.inserisci(o);
   }
   for (int k=0; k<3*dim; k++) {
      Integer i = s.estrai();
      int w = i.intValue();
      System.out.println(w);
   }
}
```

Non posso mettere un **Object** in un **Integer** …

Una Pila polimorfa …

```
public static void main(String args[]) {
   int dim = 10;
   Pila s = new Pila();
   for(int k=0; k<dim; k++){
     Integer o = new Integer(k);
     s.inserisci(o);
   }
   for (int k=0; k<3*dim; k++) {
     Integer i = (Integer) s.estrai();
     int w = i.intValue();
     System.out.println(w);
   }
}
```

Una Pila polimorfa …

```
public static void main(String args[]) {
   int dim = 10;
   Pila s = new Pila();
   for(int k=0; k<dim; k++){
      Integer o = new Integer(k);
      s.inserisci(o);
   }
   for (int k=0; k<3*dim; k++) {
      System.out.println(s.estrai());
   }
}
```

Il metodo **`toString()`** di un **`Integer`** ne stampa il valore **`int`** al suo interno

Una Pila polimorfa …

```
public static void main(String args[]) {
   int dim = 10;
   Pila s = new Pila();
   for(int k=0; k<dim; k++){
     Integer o = new Integer(k);
     s.inserisci(k);
   }
   for (int k=0; k<3*dim; k++) {
     System.out.println(s.estrai());
   }
}
```

In realtà è consentito, a partire da Java 5 …

Conversioni automatiche

collezione di `Object`

```
s.inserisci(k);
```

inserimento di un `int`

- In generale, richiede una conversione di tipo (*casting*) da **`int`** a **`Integer`** (wrapper)
- In realtà, a partire da Java 5 vengono fornite conversioni **automatiche**
  - da tipo base alla corrispondente classe wrapper (***auto-boxing***)
  - … e viceversa (***auto-unboxing***)
- Le conversioni sono realizzate dal compilatore
  - molto utili, semplificano il codice

Cosa manca?

```
int w = s.estrai();
```

`(Integer)` oppure **`(int)`**

Attenzione: non eliminano tutte le conversioni!

Conversioni forzate tra tipi riferimento: *casting*

- È possibile forzare la conversione da un tipo riferimento **`T`** ad un **sottotipo** **`T1`** purché …
- … il tipo **dinamico** dell'oggetto convertito sia un sottotipo di **`T1`**

```
Object o = new AutomobileElettrica();
Automobile a = o; // errato, Object non è un
                  // sottotipo di Automobile
Automobile a = (Automobile) o; // corretto (casting)
```

- Questa conversione esplicita viene chiamata **downcast**; quella implicita consentita dal polimorfismo viene chiamata **upcast**

Esempio

```
class A { void f1() {…} }
class B extends A { void f2() {…} }
class C extends B { void f3() {…} }
```

```
public class Test {
   …
   Test() {
      A a;
      B b = new B();
      a = b;
      a.f1();
      a.f2();
      ((B) a).f2();
      ((C) a).f3();
   }
}
```

OK: **upcast** implicito

Errore in compilazione:
"method f2 not found in class A"

OK: **downcast** esplicito

Errore a runtime:
`java.lang.ClassCastException`

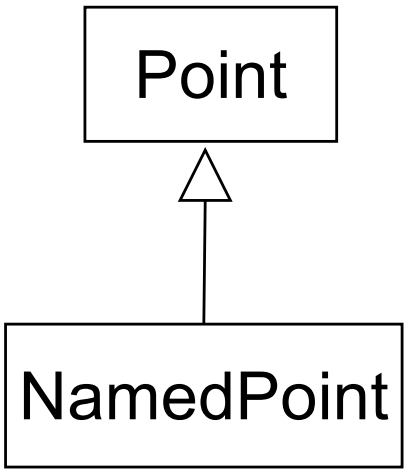
… e se invece avessimo scritto
`B b = new C();`
cosa sarebbe successo?

Determinazione del tipo: `instanceof`

- È possibile determinare il tipo dinamico di un oggetto con l'operatore **`instanceof`**
  - Utile per evitare errori di tipo a runtime dovuti a downcast

```
public static void main(String a[]){
   Point p;
   // leggi k
   if (k==1) p = new Point(2,2);
   else p = new NamedPoint(3,3,"A");
   p.getName();
   if (p instanceof NamedPoint)
      ((NamedPoint) p).getName();
}
```

di che tipo è **p**?



Ancora sulla Pila polimorfa…

```
public static void main(String args[]) {
  int dim=10;
  Pila s = new Pila();
  // inserimento valori
  for(int k=0; k<dim; k++) {
    Object o;
    if (Math.random()<0.5)
      o = new Integer(k);
    else
      o = new Float(k*Math.PI);
    s.inserisci(o);
  }
  // continua …
```

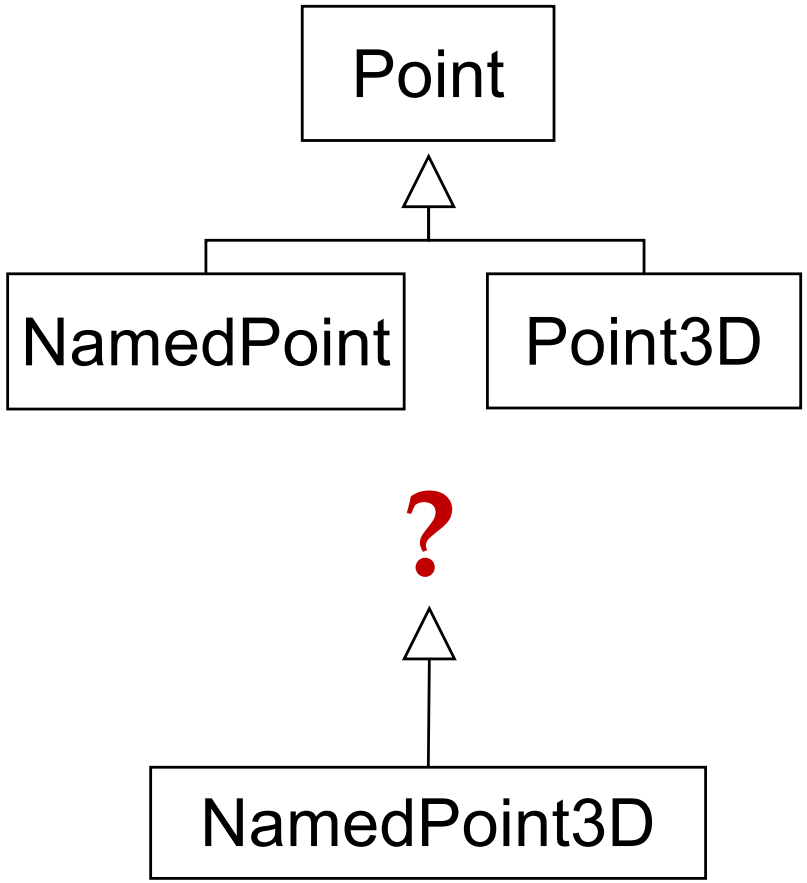
Ancora sulla Pila polimorfa…

```
    // continua … estrazione valori
    for(int k=0; k<dim; k++) {
      Object o = s.estrai();
      if (o instanceof Integer) {
        Integer i = (Integer) o;
        int w = i.intValue();
        System.out.println("an int:" + w);
      } else if (o instanceof Float) {
        Float i = (Float) o;
        float w = i.floatValue();
        System.out.println("a float:" + w);
      } else
        System.out.println("Unknown class!");
    }
  }
```

Ancora sulla Pila polimorfa…

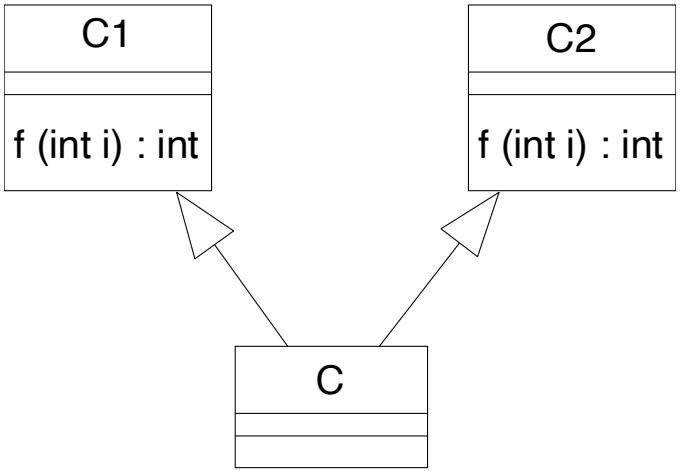
```
OUTPUT:

a float:28.274334
an int:8
an int:7
a float:18.849556
an int:5
an int:4
a float:9.424778
a float:6.2831855
a float:3.1415927
a float:0.0
```

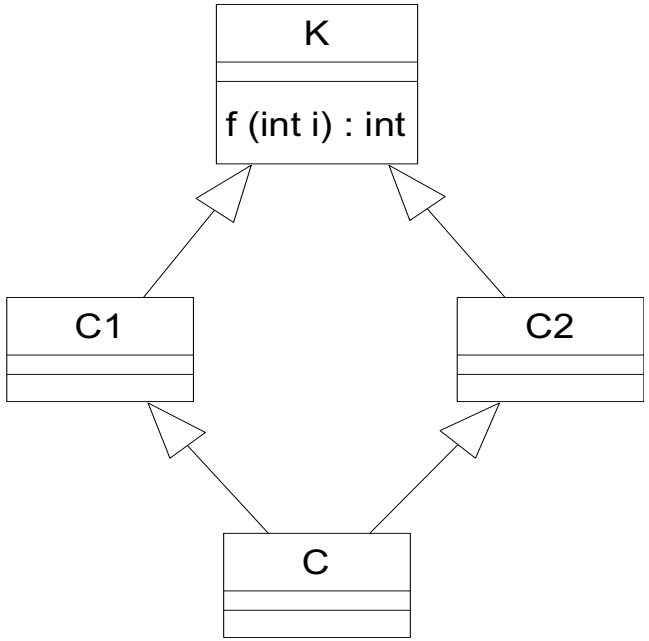
Problemi con l'ereditarietà semplice



Ereditarietà multipla: problemi

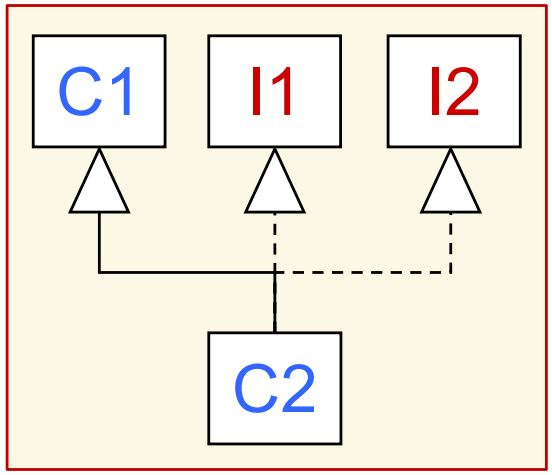
- Nei linguaggi che supportano ereditarietà multipla (es., C++) è possibile ereditare due o più metodi con la stessa firma da più superclassi…
- … il che crea un conflitto tra **implementazioni** diverse





La soluzione di Java

- Distinguere tra:
  - una gerarchia **semplice** di ereditarietà (**implementazione**)
  - una gerarchia **multipla** di specializzazione (**tipi**)



- … introducendo il costrutto delle **interfacce**
- Consente di separare l'uso dell'ereditarietà al fine di riutilizzare il codice e l'uso al fine di descrivere una gerarchia di tipi

Interfacce

- Un'interfaccia è una collezione di firme di metodi
  - Può essere vista come una classe senza attributi, i cui metodi sono tutti pubblici ed astratti
- Sintassi:

```
interface <nome> {
   <lista metodi: solo firme, senza corpo>
}
```

Da Java 8 è possibile avere «default methods» con un corpo… non considerato qui

- Un'interfaccia può altresì contenere costanti; tuttavia, in generale tale pratica è sconsigliata
- Talvolta si usa il solo nome (no metodi) per «etichettare» le classi con speciali proprietà (*tagging interfaces*)
  - Es. **`Cloneable`**, **`Serializable`**, **`Remote`**, …

Interfacce ed ereditarietà

- Una interfaccia può ereditare da **una o più** interfacce

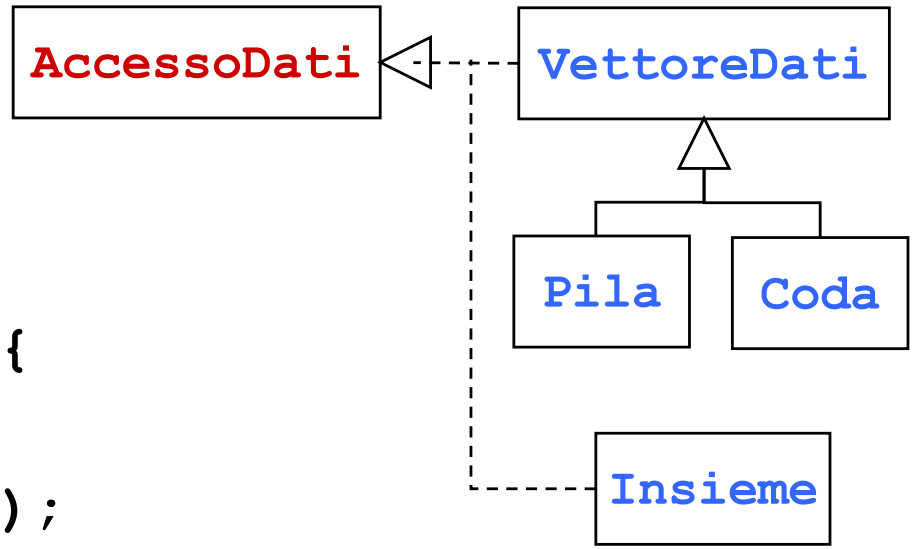
```
interface <nome> extends <nome1>,..,<nomen> {...}
```

  - La gerarchia di ereditarietà tra interfacce definisce una gerarchia di tipi
- Una classe può implementare **una o più** interfacce
  - se la classe non è astratta deve fornire un'implementazione per tutti i metodi presenti nelle interfacce che implementa; viceversa, la classe è astratta

```
class <nome> implements <nome1>,..,<nomen> {...}
```

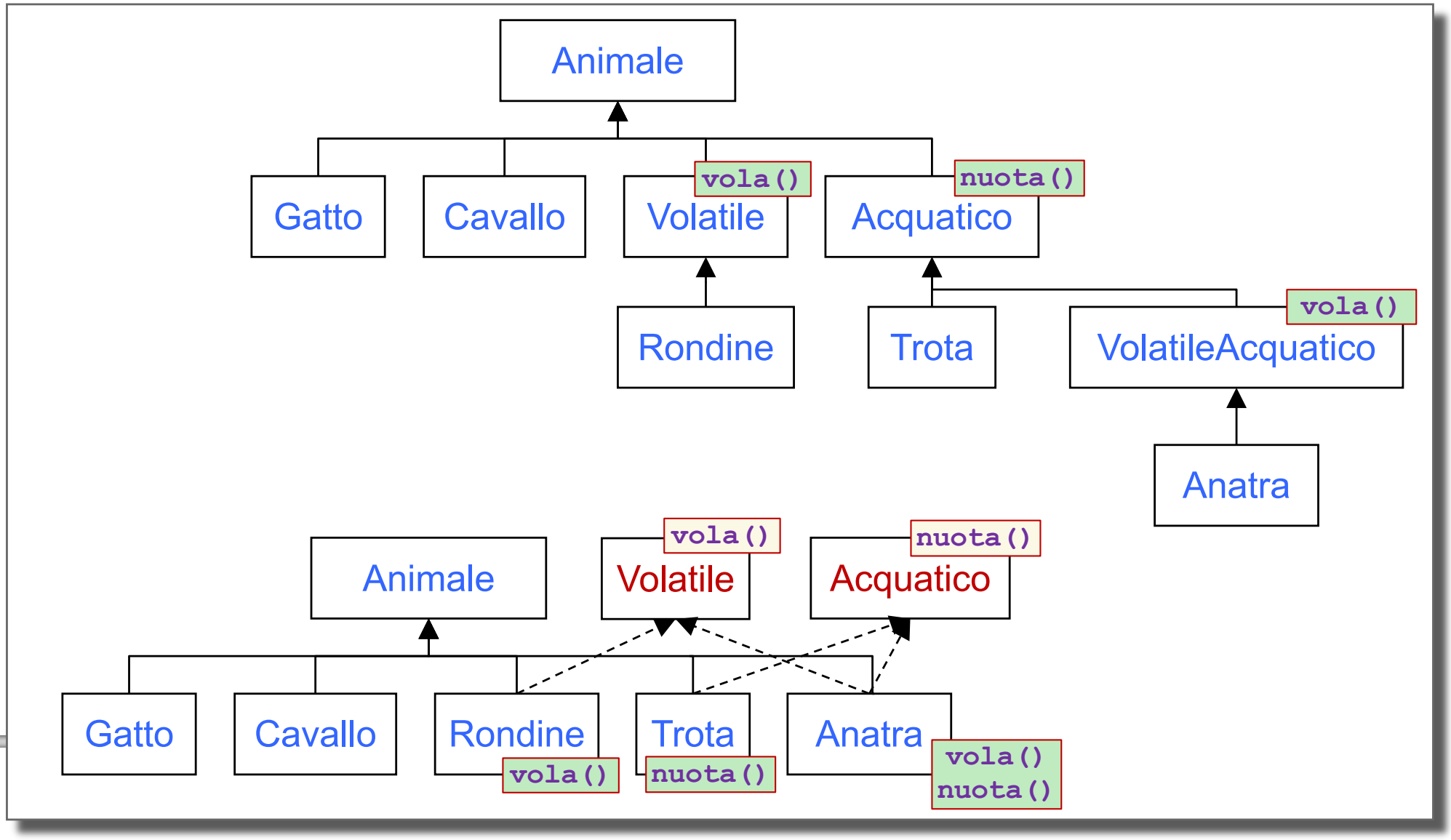
Esempio

Una classe definisce che un oggetto **è** qualcosa; un'interfaccia rappresenta i servizi (**comportamento**) che la classe deve fornire, senza tuttavia darne l'implementazione




```
package strutture;
public interface AccessoDati {
   public int estrai();
   public void inserisci(int z);
}
public abstract class VettoreDati
    implements AccessoDati { … }
public class Pila extends VettoreDati { … }
public class Coda extends VettoreDati { … }
public class Insieme implements AccessoDati {…}
```

Esempio


Animale
Gatto
Cavallo
Volatile
vola()
Acquatico
nuota()
Rondine
Trota
VolatileAcquatico
vola()
Anatra
Animale
Volatile
vola()
Acquatico
nuota()
Gatto
Cavallo
Rondine
vola()
Trota
nuota()
Anatra
vola()
nuota()

Polimorfismo ed interfacce

- Una interfaccia può essere utilizzata per definire il tipo di una variabile
- Valgono le regole del polimorfismo:
  tale variabile potrà riferirsi ad un qualsiasi oggetto che implementi l'interfaccia

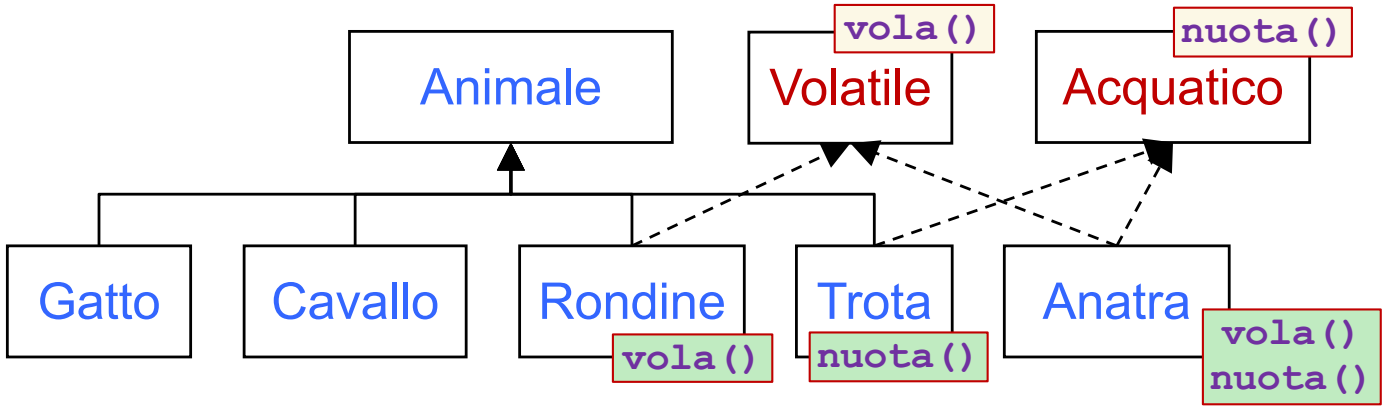
```
AccessoDati o = new Pila();
o.inserisci(5);
```

- Ma un'interfaccia non può essere usata per creare un oggetto!

```
AccessoDati o =
   new AccessoDati();
```

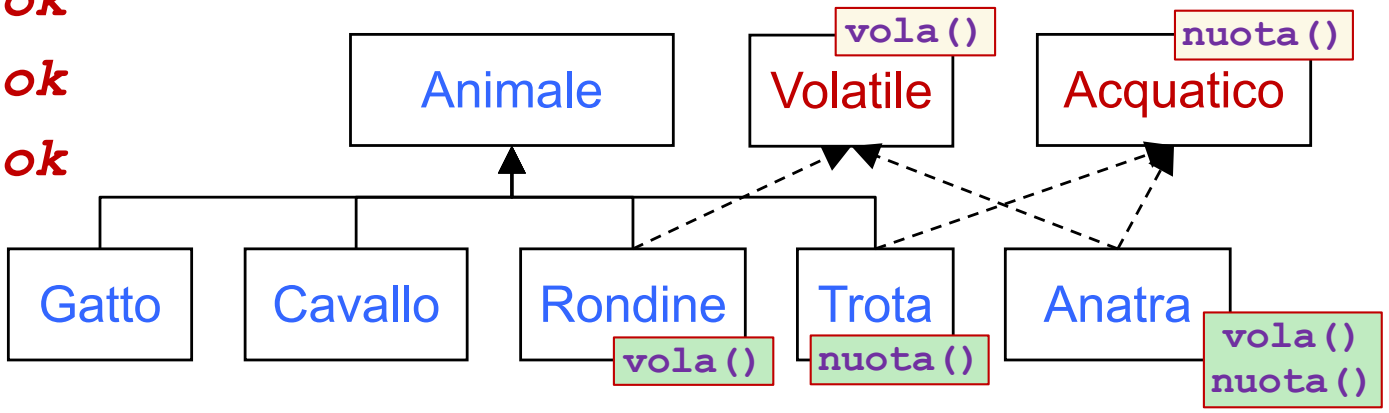
Esempio

```
Animale g = new Gatto();
Acquatico t = new Trota();
Anatra a = new Anatra();
Acquatico c = new Acquatico();
Volatile l = g;
Volatile v = a;
Acquatico q = a;
g.vola();
v.vola();
t.nuota();
a.nuota();
```



Esempio

```
Animale g = new Gatto();                    // ok
Acquatico t = new Trota();                  // ok
Anatra a = new Anatra();                    // ok
Acquatico c = new Acquatico();              // errato
Volatile l = g;     // errato
Volatile v = a;     // ok
Acquatico q = a;    // ok
g.vola();   // errato
v.vola();   // ok
t.nuota();  // ok
a.nuota();  // ok
```



Ereditarietà, polimorfismo, e array

- Se **`X`** è una superclasse di **`Y`** allora l'array **`X[]`** è "super-array" dell'array **`Y[]`**
  - Lo stesso vale per gli array multidimensionali
- **Tale scelta non è type safe**

```
void f(X[] ax) {
  ax[0] = new X();
}
...
f(new Y[10]);
```