

Information hiding:
visibilità

Information hiding in Java

- La visibilità degli attributi e metodi di una classe **C** può essere:
 - **public**
 - visibili a tutti
 - vengono ereditati
 - **protected**
 - visibili solo alle sottoclassi
 - vengono ereditati
 - **«package» (nessun modificatore specificato)**
 - visibili solo alle classi dichiarate nel package di **C**
 - **private**
 - visibili solo all'interno della stessa classe
 - non visibili nelle sottoclassi

Modificatori di visibilità

	visibilità			
modificatore	classe	package	sottoclasse	mondo
private	Y	N	N	N
“package”	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

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

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

Static e dynamic binding

- Il C++ offre al programmatore complessi meccanismi per decidere se usare **dynamic binding** (tipo deciso a runtime) o **static binding** (tipo deciso a compile time)
 - In C++ la keyword “virtual” abilita il dynamic binding, che altrimenti è sempre static
- In Java le decisioni sono sempre a runtime ... salvo quando sia possibile decidere automaticamente a compile time, e cioè per:
 - metodi **private**, **static** e **final**
 - costruttori

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...):

```
Test2.java:13: error: f1() in Derived cannot override f1() in Test2  
    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();  
    }  
    public 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 subclass

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)

Una Pila polimorfa ...

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

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 ...

Nella class **Pila**

abilita lo static binding

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

Una Pila polimorfa ...

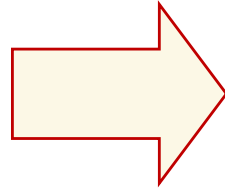
```
public static void main(String args[]) {  
    int dim = 10;  
    Pila s = new Pila(dim);  
    for(int k=0; k<dim; k++){  
        s.inserisci(k);  
    }  
    for (int k=0; k<3*dim; k++) {  
        System.out.println(s.estraini());  
    }  
}
```

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

Integer

int

Float

float

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

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

Interface



Interfacce

Un'interfaccia è una classe completamente astratta, senza attributi (solo una collezione di firme di metodi pubblici e astratti)

Sintassi:

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

Un'interfaccia può contenere costanti.

Talvolta si usano interfacce completamente vuote (senza metodi) per «etichettare» classi con speciali proprietà (*tagging interfaces*)

Es. **Cloneable**, **Serializable**, **Remote**, ...

Interfacce ed ereditarietà

Una interfaccia può ereditare da **una o più** interfacce (ma non da classi!)

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

Interfacce ed ereditarietà

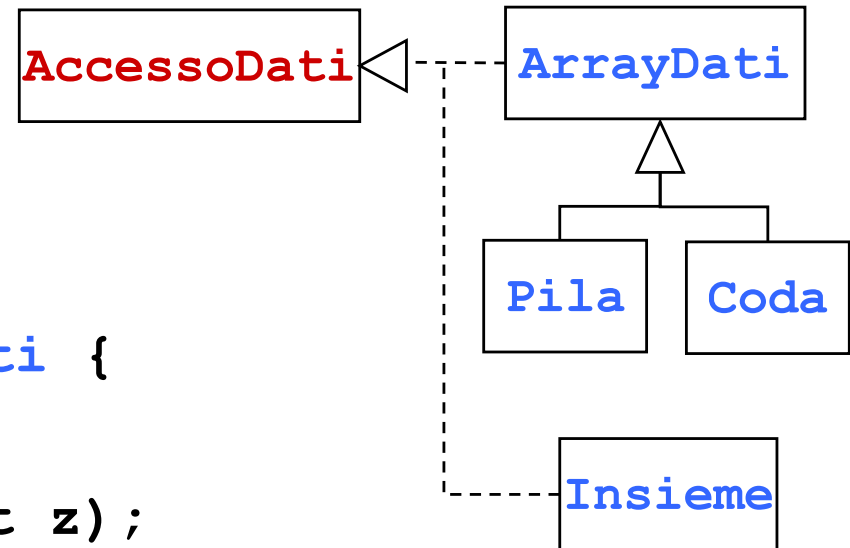
Una classe può implementare **una o più** interfacce, e DEVE implementarne tutti i metodi (a meno che non sia astratta)

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


Una classe definisce che un oggetto **è** qualcosa; un'interfaccia rappresenta i servizi (**comportamento**) che la classe deve fornire

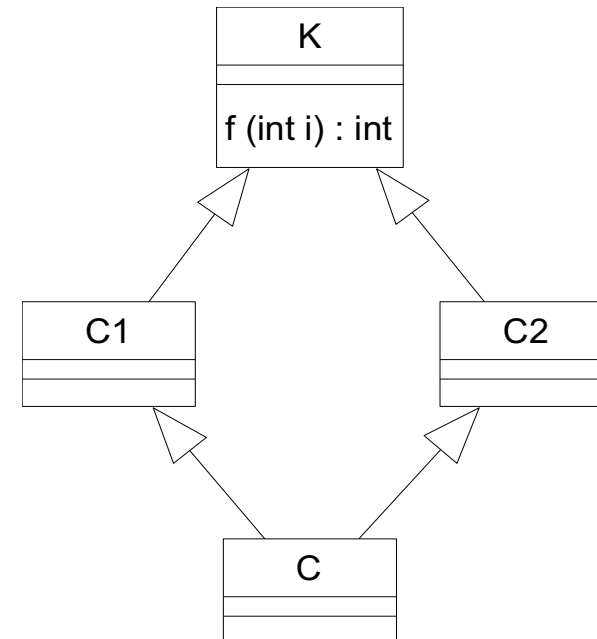
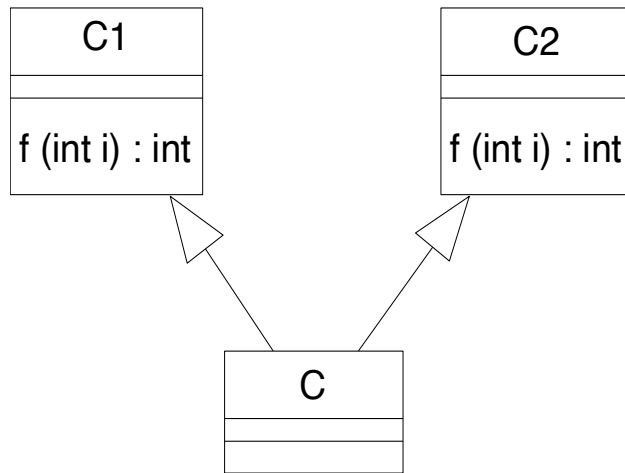
Esempio

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

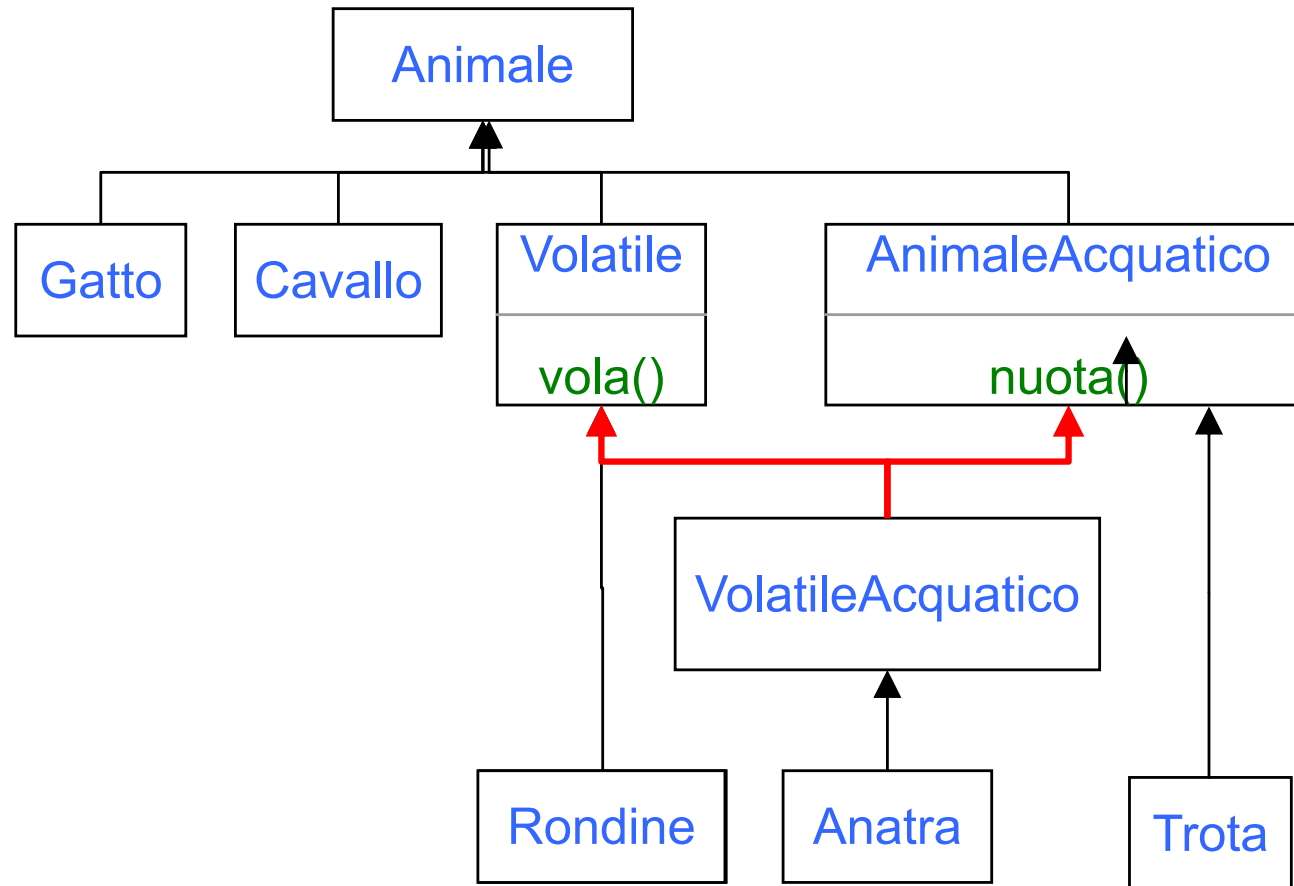


interface resolve i problemi della multiple inheritance

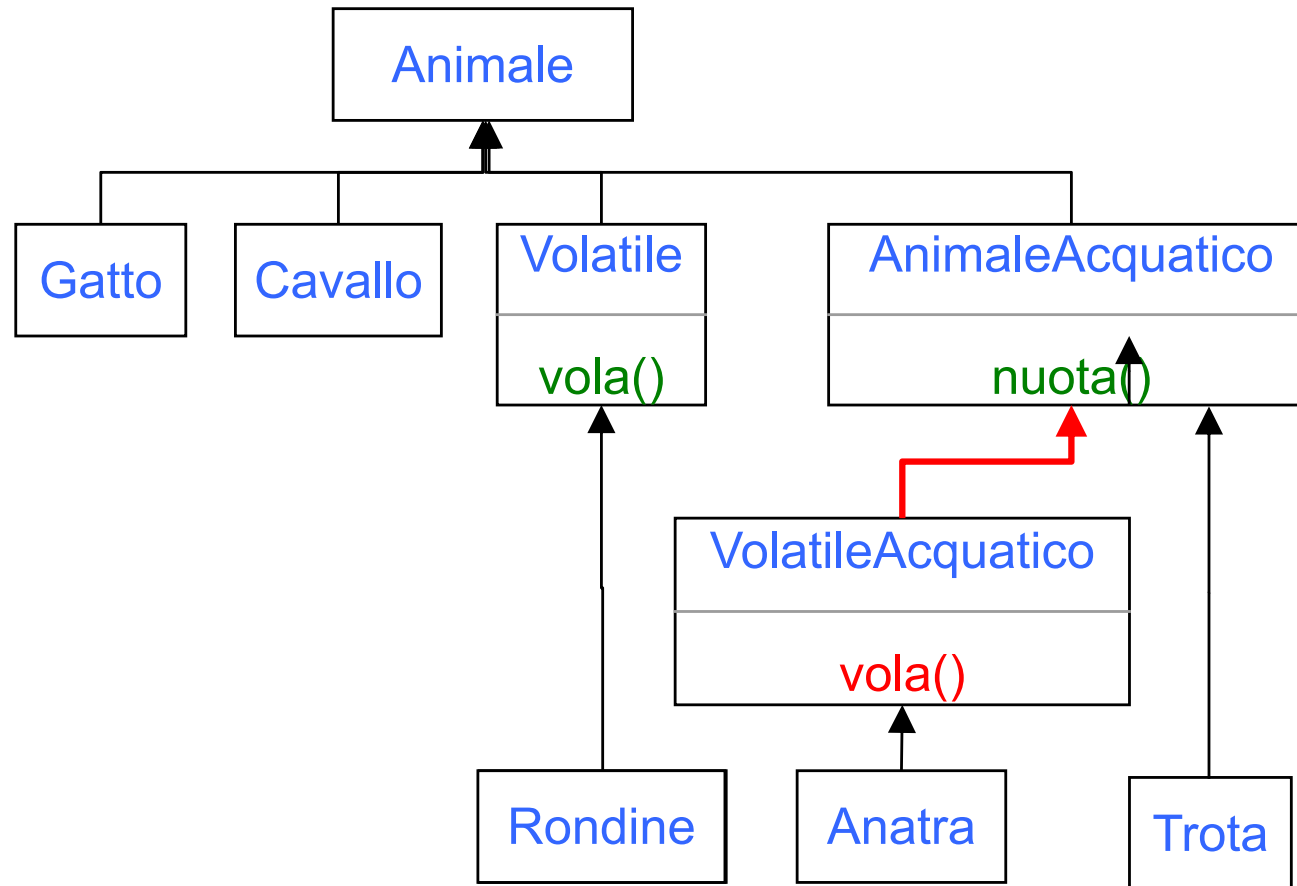
Nei linguaggi che supportano ereditarietà multipla (es., C++) è possibile ereditare due o più metodi con la stessa firma da più superclassi...
... il che crea un conflitto tra **implementazioni** diverse



Esempio: senza interfacce, eredità multipla

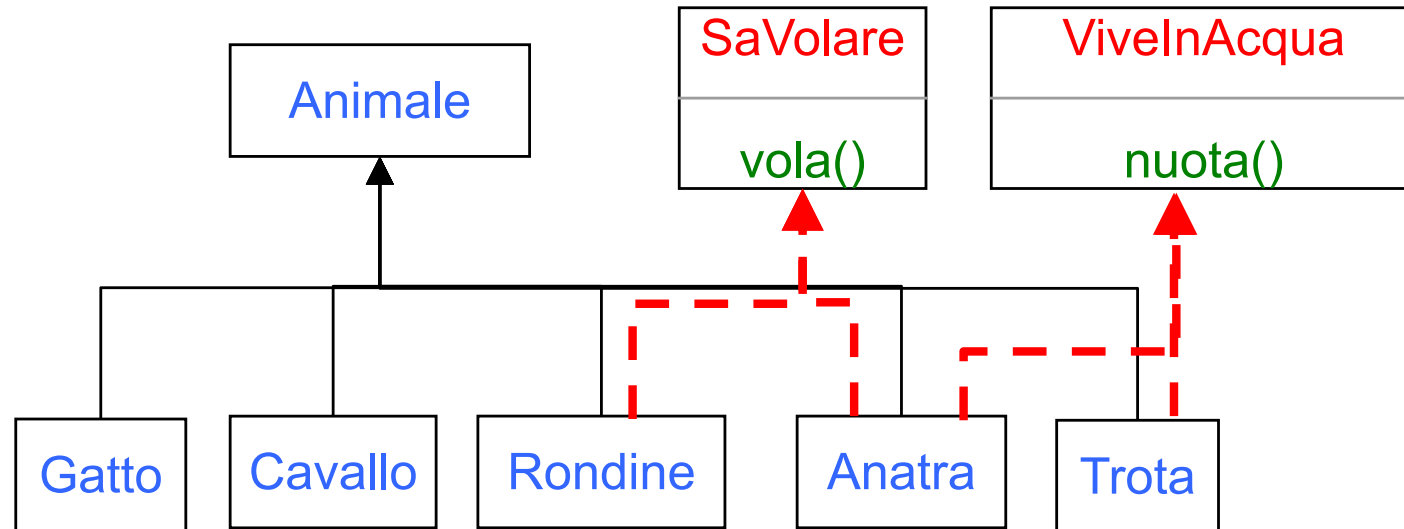


Esempio: senza interfacce, eredità singola



Approccio tassonomico

Esempio: con le interfacce e multiple inheritance



“la rondine è un animale che sa volare”

“l’anatra è un animale che sa volare e vive in acqua”

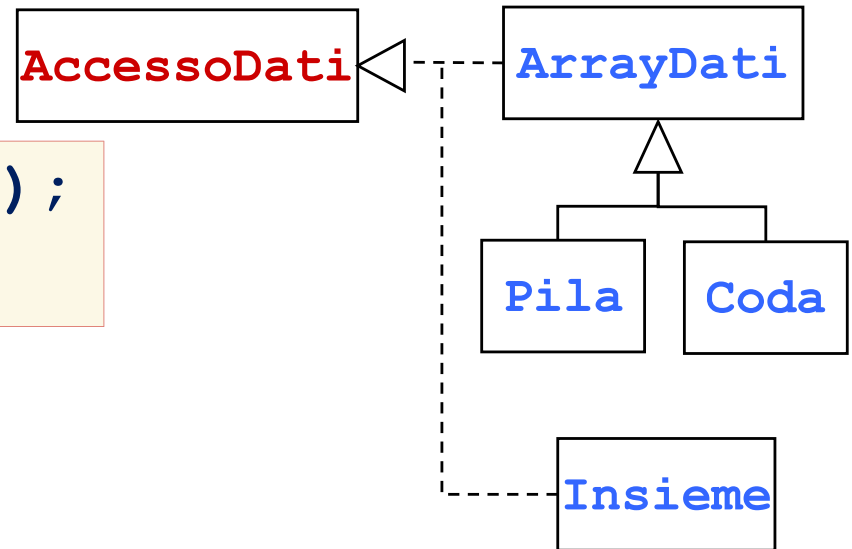
Polimorfismo ed interfacce

Una interfaccia può essere utilizzata per definire il tipo di una variabile

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

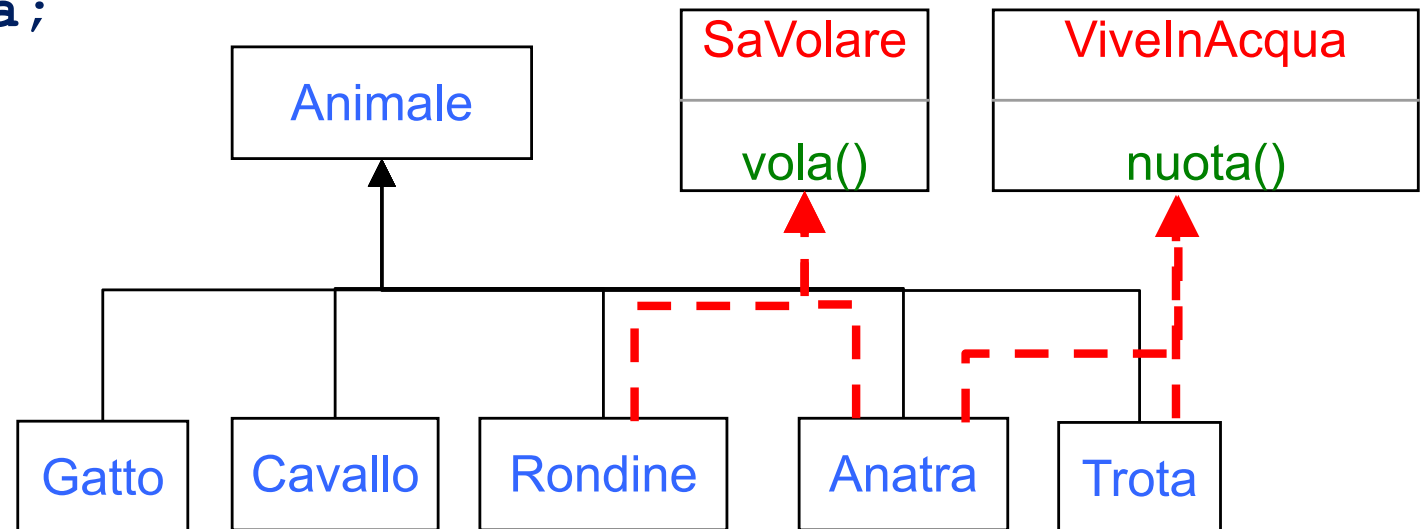
Ma un'interfaccia non può essere usata per creare un oggetto!

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



Esercizio: quali di queste sono errate?

```
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();  
t.vola();  
a.nuota();
```



Soluzione

```
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
                                t.vola(); // errato
a.nuota();                         // ok
```