



## Classe Coin: Sovrascrivere equals

---

```
public boolean equals(Object otherObject) {  
    if (otherObject == null) return false;  
    if (getClass() != otherObject.getClass())  
        return false;  
    Coin other = (Coin)otherObject;  
    return name.equals(other.name)  
        && value == other.value;  
}
```



## Sottoclassi: Sovrascrivere equals

---

- Creiamo una sottoclasse di **Coin**: **CollectibleCoin**
  - Una moneta da collezione è caratterizzata dall'anno di emissione (vbl. istanza aggiuntiva)

```
public CollectibleCoin extends Coin{  
    ...  
    private int year;  
}
```

- Due monete da collezione sono uguali se hanno uguali nomi, valori e anni di emissione
  - Ma name e value sono variabili private della superclasse!
  - Il metodo equals della sottoclasse non può accedervi



## Sottoclassi: Sovrascrivere `equals`

---

- Soluzione: il metodo `equals` della sottoclasse invoca il metodo omonimo della superclasse
  - Se il confronto ha successo, procede confrontando le altre vbl aggiuntive

```
public boolean equals(Object otherObject) {  
    if (!super.equals(otherObject)) return false;  
  
    CollectibleCoin other =  
        (CollectibleCoin) otherObject;  
    return year == other.year;  
}
```



## Sovrascrivere `clone`

---

- Il metodo `clone` della classe `Object` crea un nuovo oggetto con lo stesso stato di un oggetto esistente (copia profonda o clone)
  - `protected Object clone()`
- Se `x` è l'oggetto che vogliamo clonare, allora
  - `x.clone()` e `x` sono oggetti con diversa identità
  - `x.clone()` e `x` hanno lo stesso contenuto
  - `x.clone()` e `x` sono istanze della stessa classe



## Sovrascrivere `clone`

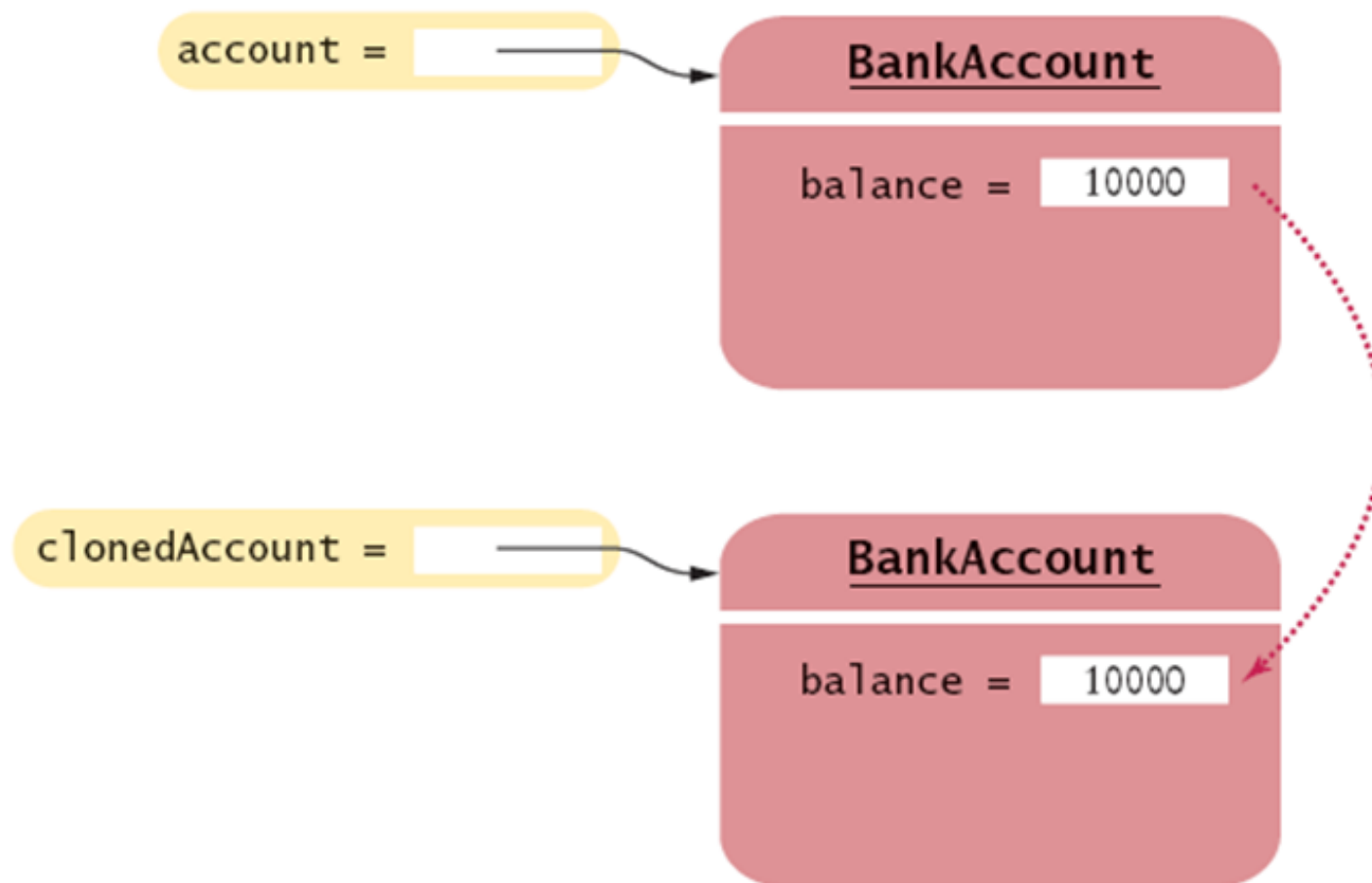
---

- Clonare un conto corrente

```
public Object clone()  
{  
    BankAccount cloned= new BankAccount();  
    cloned.balance = balance;  
    return cloned;  
}
```

# Clonare Oggetti

---





# Sovrascrivere `clone`

---

- Il tipo restituito dal metodo `clone` è **Object**
- Se invochiamo il metodo dobbiamo usare un cast per dire al compilatore che **`account1.clone()`** ha lo stesso tipo di **`account2`**:

```
BankAccount account1 = . . . ;  
BankAccount account2 =  
    (BankAccount) account1.clone() ;
```



# L' ereditarietà e il metodo clone

---

- Abbiamo visto come clonare un oggetto **BankAccount**

```
public Object clone() {  
    BankAccount cloned= new BankAccount();  
    cloned.balance = balance;  
    return cloned;  
}
```

- Problema: questo metodo non funziona nelle sottoclassi!

```
SavingsAccount s= new SavingsAccount(0.5);  
Object clonedAccount = s.clone(); //NON VA BENE
```





## L' ereditarietà e il metodo `clone`

---

- Viene costruito un conto bancario e non un conto di risparmio!
  - **SavingsAccount** ha una variabile aggiuntiva, che non viene considerata!
- Possiamo invocare il metodo `clone` della classe **Object**
  - Crea un nuovo oggetto dello stesso tipo dell'oggetto originario
  - Copia le variabili di istanza dall'oggetto originario a quello clonato



## L' ereditarietà e il metodo clone

---

```
public class BankAccount{  
    ...  
    public Object clone() {  
        ...  
        //invoca il metodo Object.clone()  
        Object cloned = super.clone();  
        return cloned;  
    }  
}
```



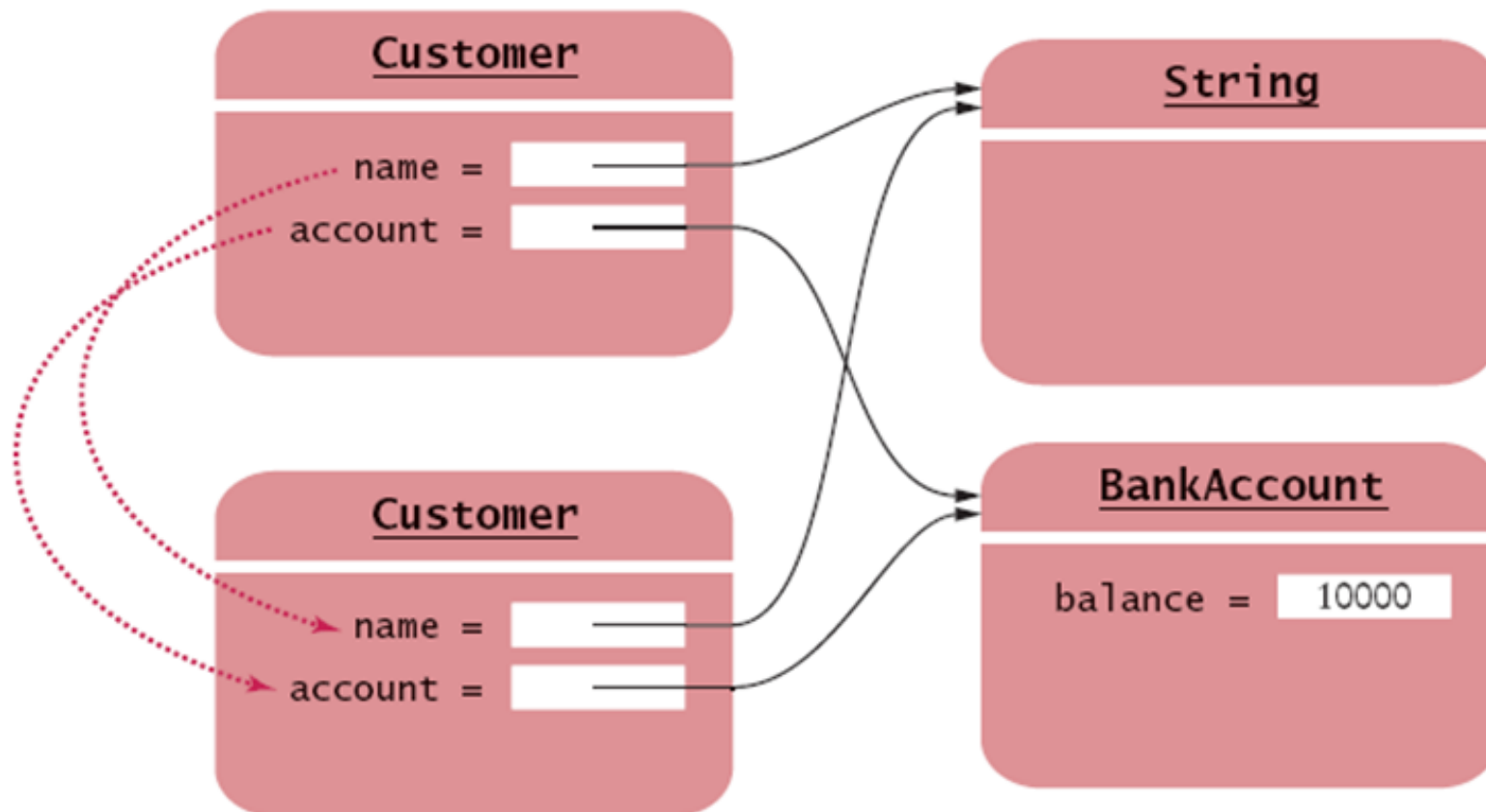
# L' ereditarietà e il metodo `clone`

---

- Consideriamo una classe **Customer**
  - Un cliente è caratterizzato da un nome e un conto corrente
- L' oggetto originale e il clone condividono un oggetto di tipo **String** e uno di tipo **BankAccount**
  - Nessun problema per il tipo **String** (oggetto immutabile)
  - Ma l'oggetto di tipo **BankAccount** potrebbe essere modificato da qualche metodo di **Customer**!
  - Andrebbe clonato anch'esso

# L' ereditarietà e il metodo clone

- Problema: viene creata una copia superficiale
  - Se un oggetto contiene un riferimento ad un altro oggetto, viene creata una copia di riferimento all'oggetto, non un clone!





## L' ereditarietà e il metodo `clone`

---

- Il metodo `Object.clone` si comporta bene se un oggetto contiene
  - Numeri, valori booleani, stringhe
- Bisogna però usarlo con cautela se l'oggetto contiene riferimenti ad altri oggetti
  - Quindi è inadeguato per la maggior parte delle classi!



# L' ereditarietà e il metodo `clone`

---

- Precauzioni dei progettisti di Java:
  - Il metodo `Object.clone` è stato dichiarato protetto
    - Non possiamo invocare `x.clone()` se non all'interno della classe, di una sottoclasse o dello stesso pacchetto dell'oggetto `x`
  - Una classe che voglia consentire di clonare i suoi oggetti deve implementare l'interfaccia `Cloneable`
    - In caso contrario viene lanciata un'eccezione di tipo `CloneNotSupportedException`
    - Tale eccezione va catturata anche se la classe implementa `Cloneable`
- In genere, quando sovrascriviamo `clone` lo ridefiniamo `public` così è possibile usarlo dovunque.



# L' interfaccia Cloneable

---

```
public interface Cloneable{  
}
```

- Interfaccia contrassegno
  - Non ha metodi
  - Usata solo per verificare se un'altra classe la realizza
  - Se l'oggetto da clonare non è un esemplare di una classe che la realizza viene lanciata l'eccezione



# Clonare un BankAccount

---

```
public class BankAccount implements Cloneable
{
    ...

    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            //non succede mai perché implementiamo Cloneable
            return null;
        }
    }
}
```





# Clonare un Customer

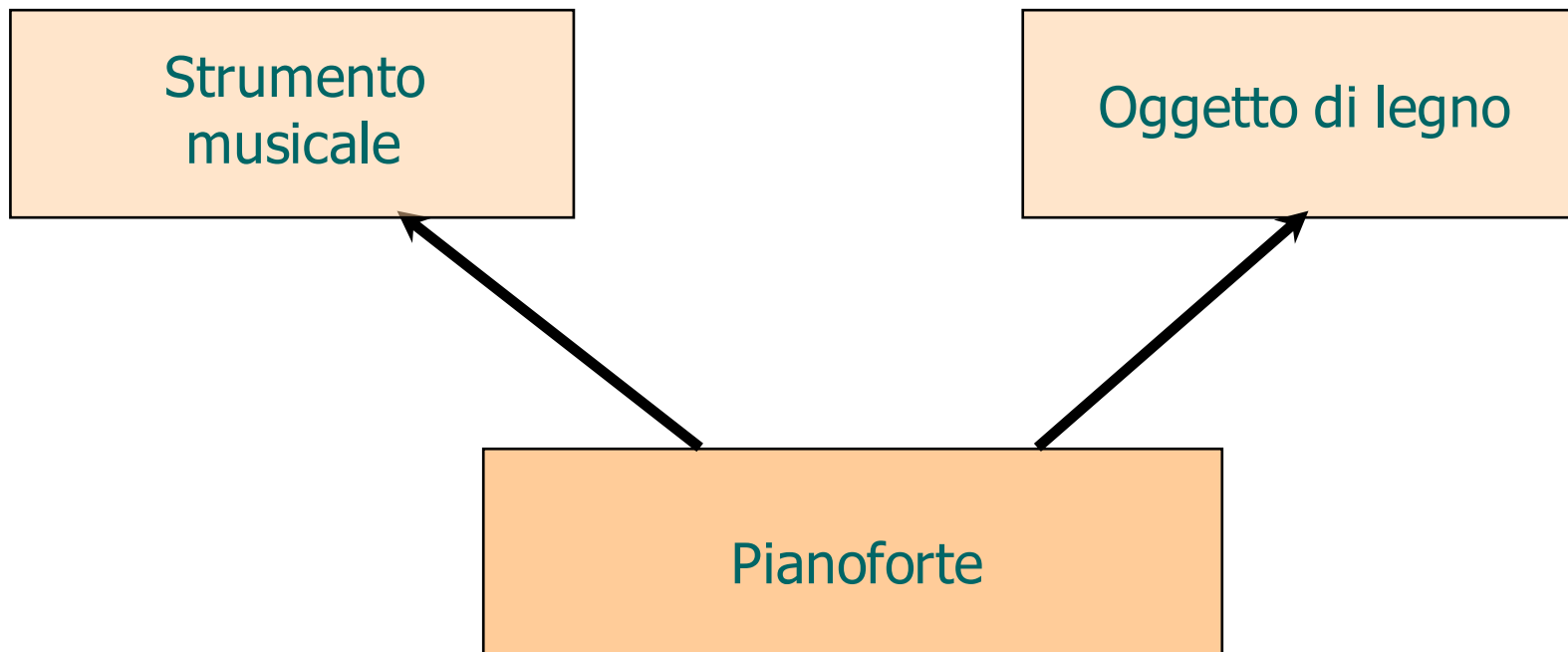
---

```
public class Customer implements Cloneable
{
...
    public Object clone()
    {
        try
        {
            Customer cloned = (Customer)super.clone();
            cloned.account = (BankAccount)account.clone();
            return cloned;
        }
        catch (CloneNotSupportedException e)
        {
            //non succede mai perché implementiamo Cloneable
            return null;
        }
    }
    private String name;
    private BankAccount account;
}
```

# Ereditarietà multipla

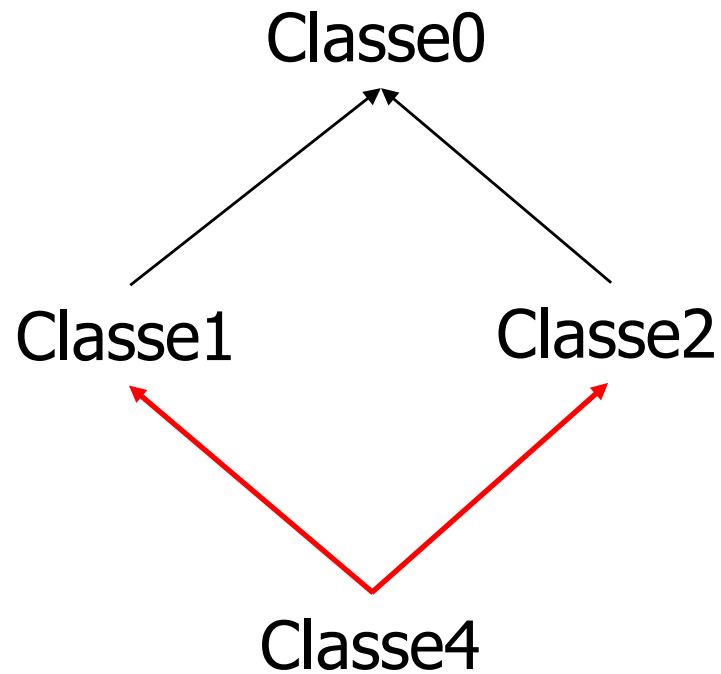
---

- Una classe può avere più padri di pari livello
- In Java non è consentita, per la fragilità del meccanismo
- Realizzata attraverso il concetto di interfaccia.

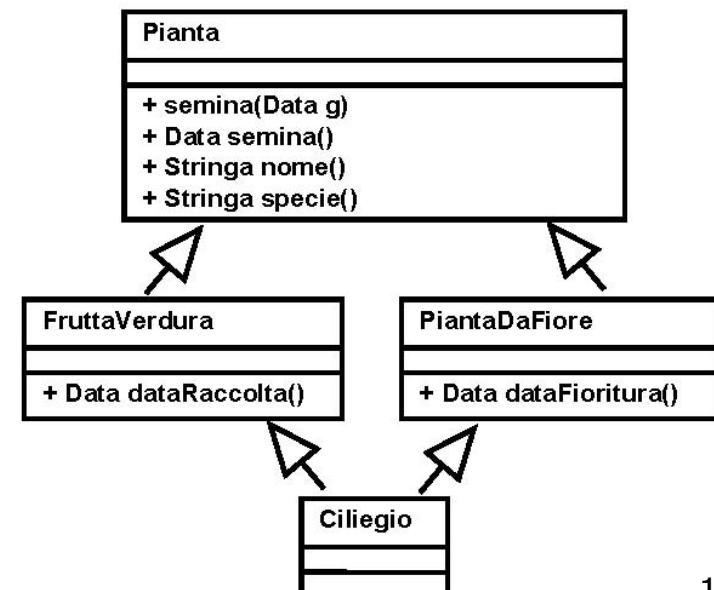
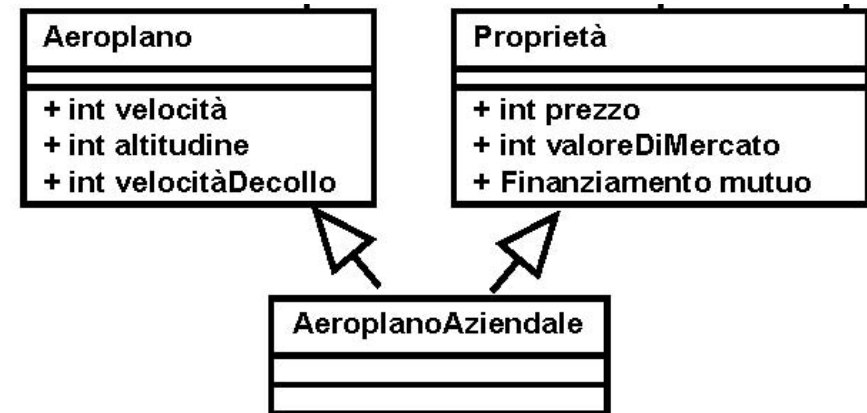
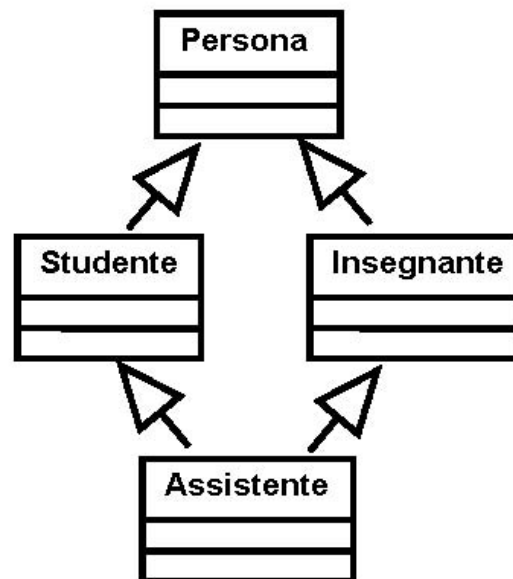
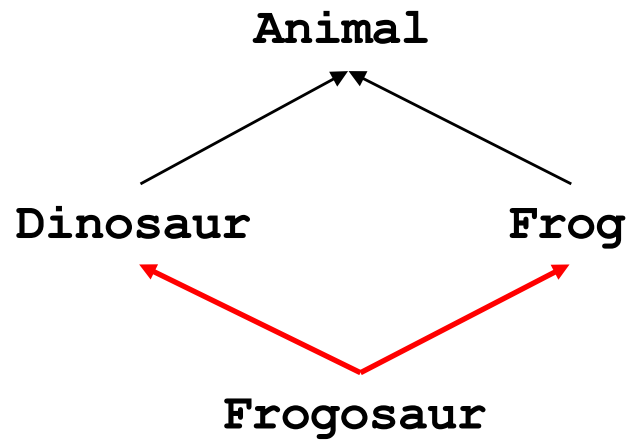


# Problemi con l'ereditarietà multipla: *l'ereditarietà a diamante*

---



# Esempi





# La gerarchia del frogosauro

---

```
class Animal {
    void talk() {
        System.out.println("...");
    }
}

class Frog extends Animal {
    void talk() {
        System.out.println("Ribit, ribit.");
    }
}

class Dinosaur extends Animal {
    void talk() {
        System.out.println("I'm a dinosaur: I'm OK!");
    }
}
```



## Il frogosauo

---

```
// (non compila.)  
class Frogosaur extends Frog, Dinosaur {  
}
```

○ Cosa dovrebbe fare la seguente chiamata a `talk()`?

```
Animal animal = new Frogosaur();  
animal.talk();
```



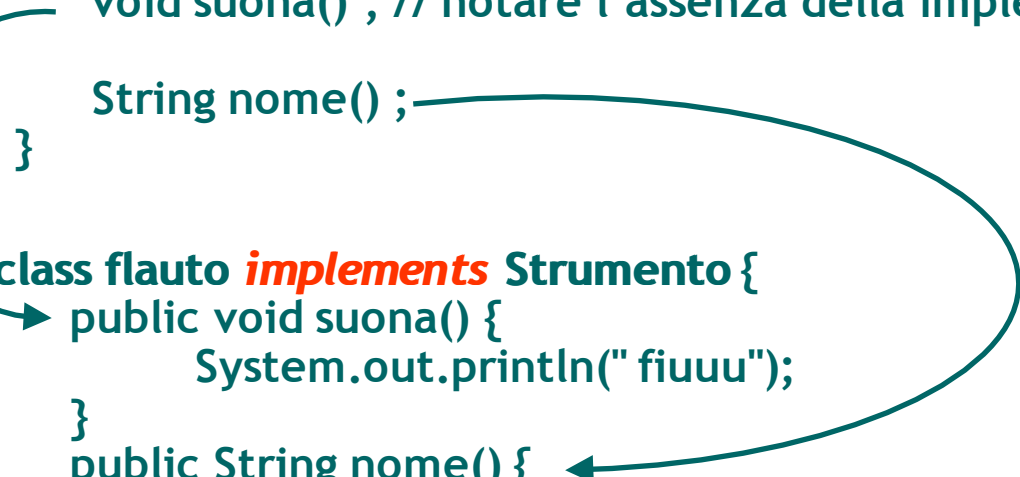
# Problemi con l'Ereditarietà multipla

---

- Se un membro (metodo o campo) è definito in entrambe le classi base, da quale delle due la classe estesa "eredita"?
- E se le due classi base a loro volta estendono una superclasse comune ?
- Il problema è legato alle implementazioni (che vengono ereditate)
- Per questo motivo:
  - una classe può implementare tante interfacce
  - ma può estendere una sola classe

# Interfacce: esempio (8)

```
interface Strumento {  
    String descrizione = "azioni di uno strumento musicale generico";  
    // valore implicitamente static e final  
  
    void suona() ; // notare l'assenza della implementazione  
  
    String nome() ;  
}  
  
class flauto implements Strumento {  
    public void suona() {  
        System.out.println(" fiuuu");  
    }  
    public String nome() {  
        return "flauto";  
    }  
    public String descrizione() {  
        return "sono un flauto di marca";  
    }  
}
```





# Esempio di interfacce

```
interface Strumento  
{  
    suona ( );  
    nome ( );  
}
```

```
interface InVendita {  
    prezzo ( );  
    disponibile ( );  
}
```

Oggetto di legno

Extends

Pianoforte In Vendita

Implements

Implements



# Interfacce

---

- Una classe può estendere una sola classe ma può implementare un numero illimitato di interfacce
- ```
class PianoforteInVendita extends  
OggettoDiLegno implements Strumento,  
InVendita {  
    ...  
}
```
- Un *pianoforte in vendita* è un oggetto di legno **ed** uno strumento musicale è un bene in vendita