



Fattorizzazione

- Abbiamo visto l'ereditarietà usata per estendere le funzionalità di una classe
- L'ereditarietà può essere anche usata per spostare un comportamento comune a due o più classi in una singola superclasse
- Un esempio: un sistema di inventario
 - Obiettivi (Lens)
 - Pellicole (Films)
 - Macchine fotografiche (Cameras)



Proprietà

- Lens
 - Focal length
 - Zoom/ fixed lens
- Film
 - Recommended storage temperature
 - Film speed
 - Number of exposures
- Camera
 - Lens included?
 - Maximum shutter speed
 - Body color



Proprietà di tutti gli item inventariati

- Description
- Inventory ID
- Quantity on hand
- Price



Classe Lens

```
class Lens {  
    Lens(...) {...} // Constructor  
    String getDescription() {return description;};  
    int getQuantityOnHand() {return quantityOnHand;}  
    int getPrice() {return price;}  
    ...  
    // Methods specific to Lens class  
    ...  
    String description;  
    int inventoryNumber;  
    int quantityOnHand;  
    int price;  
    boolean isZoom;  
    double focalLength;  
}
```




Classe Film

```
class Film {  
    Film(...) {...} // Constructor  
    String getDescription() {return description;};  
    int getQuantityOnHand() {return quantityOnHand;}  
    int getPrice() {return price;}  
    ...  
    // Methods specific to Film class  
    ...  
    String description;  
    int inventoryNumber;  
    int quantityOnHand;  
    int price;  
    int recommendedTemp;  
    int numberOfExposures;  
}
```



Classe Camera

```
class Camera {  
    Camera(...) {...} // Constructor  
    String getDescription() {return description;};  
    int getQuantityOnHand() {return quantityOnHand;}  
    int getPrice() {return price;}  
    ...  
    // Methods specific to Camera class  
    ...  
    String description;  
    int inventoryNumber;  
    int quantityOnHand;  
    int price;  
    boolean hasLens;  
    int maxShutterSpeed;  
    String bodyColor;  
}
```



Estrazione di un comportamento comune e fattorizzazione in una superclasse

- Notare la ridondanza nelle tre classi precedenti
- Ogni classe in realtà sta modellando due entità
 - Un generico inventory item
 - Uno specifico item - lens, film, camera
- Ricordate, OOP è anche responsibility-driven programming!!
- Dividere le responsabilità



La superclasse

```
class InventoryItem {  
    InventoryItem(...) {...}  
    String getDescription() {...}  
    int inventoryID() {...}  
    int getQtyOnHand() {...}  
    int getPrice() {...}  
  
    String description;  
    int inventoryNumber;  
    int qtyOnHand;  
    int price;  
}
```




Le tre sottoclassi

- Il codice della classe Lens

```
class Lens extends InventoryItem {  
    Lens(...) {...}  
    ...  
    // Methods specific to Lens class  
    ...  
    boolean isZoom;  
    double focalLength;  
}
```

- In maniera analoga per Film e Camera



Lavorare con la gerarchia di classi

```
InventoryItem [] invarr = new InventoryItem[ 3];  
invarr[0] = new Lens(...);  
invarr[1] = new Film(...);  
invarr[2] = new Camera(...);  
for (int i = 0; i < invarr.length; i++)  
    System.out.println(invarr[i].getDescription() + ": " +  
        invarr[i].getQtyOnHand() +  
        "available");
```



Accedere ai dati delle sottoclassi

- Un metodo print nella superclasse è capace di visualizzare solo gli elementi comuni della superclasse
- Come visualizzare i dati dei singoli oggetti ?
 - *focal length* e *zoom* per lens
 - *speed* e *temperature* per film
- InventoryItem non conosce queste proprietà!!



Usare il polimorfismo

- Aggiungere un metodo print a ogni sottoclasse

```
class Lens extends InventoryItem {  
    ...  
    void print() {  
        // prints Lens- specific data  
    }  
    ...  
}
```
- Allo stesso modo per Film e Camera ...
- Ora definiamo un metodo print nella superclasse

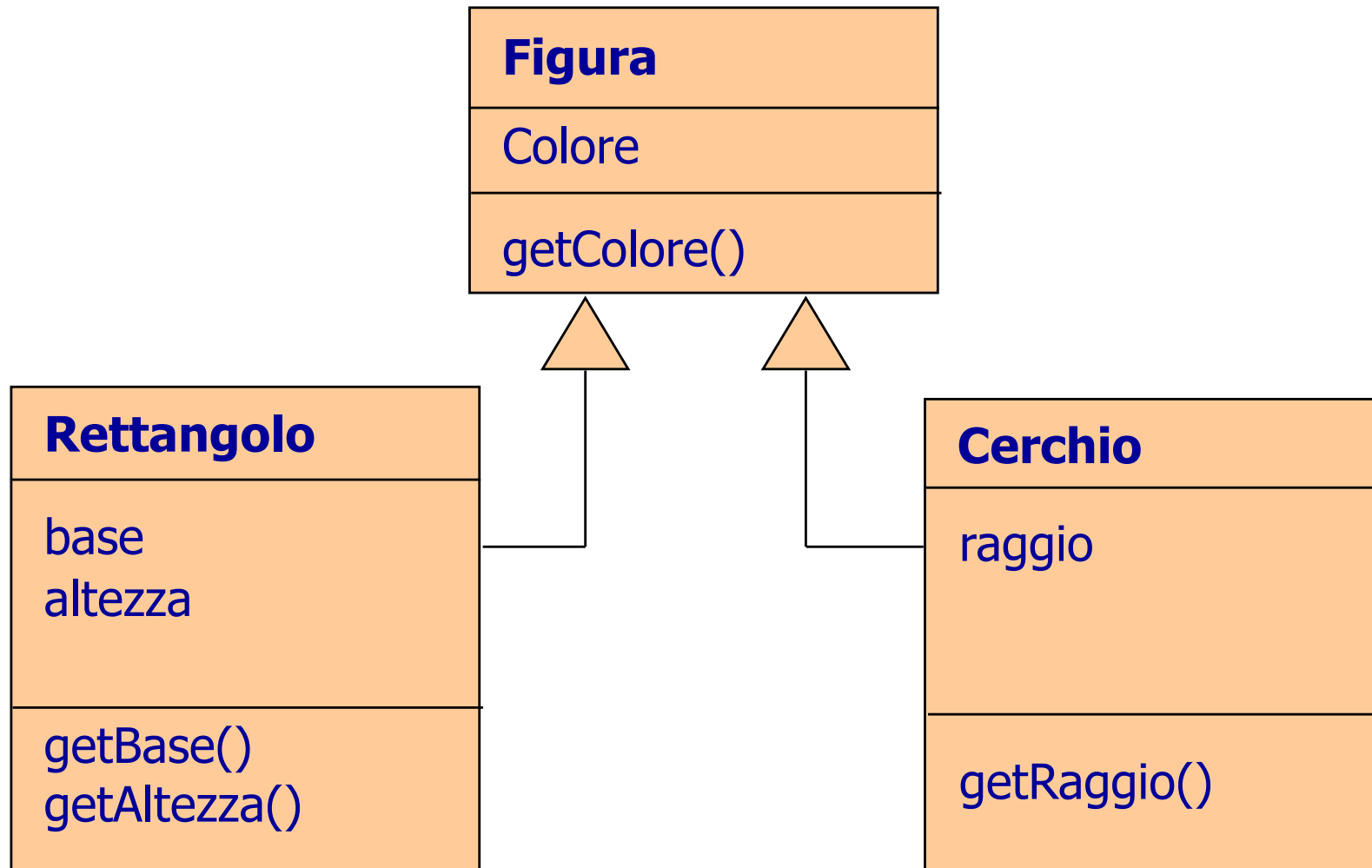
```
class InventoryItem {  
    ...  
    void print() {...}  
    ...  
}
```



Inizializzazione degli oggetti

- Costruttore di default
 - E' l'inizializzazione eseguita automaticamente se non sono stati definiti altri costruttori
- Il costruttore di una sottoclasse può chiamare quello della superclasse tramite il metodo *super*
 - La chiamata a super deve essere la prima istruzione del costruttore

Ereditarietà e riuso del codice



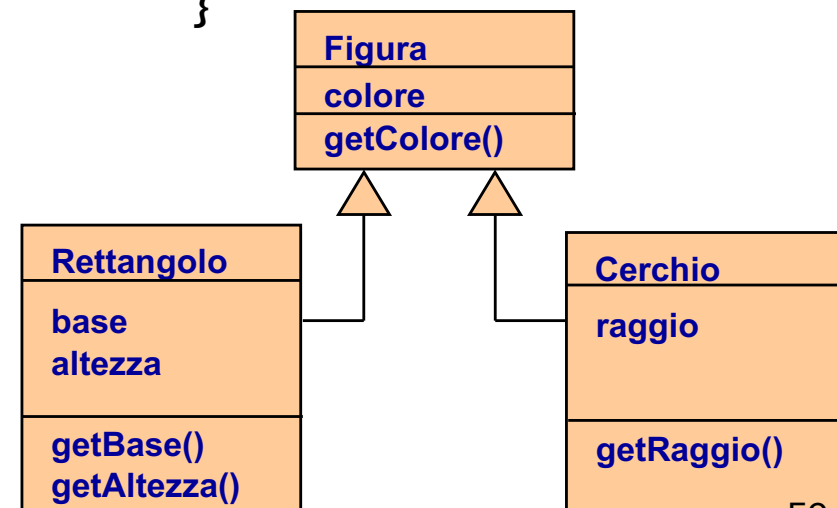
Esempio (2)

```
class Figura{
    private String colore;
    Figura(String col) {
        colore = col;
    }
    String getColore() {
        return colore;
    }
}
```

```
class Rettangolo extends Figura {
    private double altezza;
    private double base;
    Rettangolo (String col, double alt,
                double bas) {

        super(col);
        altezza = alt;
        base = bas;
    }
    double getArea() {
        return altezza * base;
    }
}
```

```
class Cerchio extends Figura {
    private double raggio;
    Cerchio (String col, double rag) {
        super(col);
        raggio = rag;
    }
    double getArea() {
        return raggio * raggio * 3.14;
    }
}
```

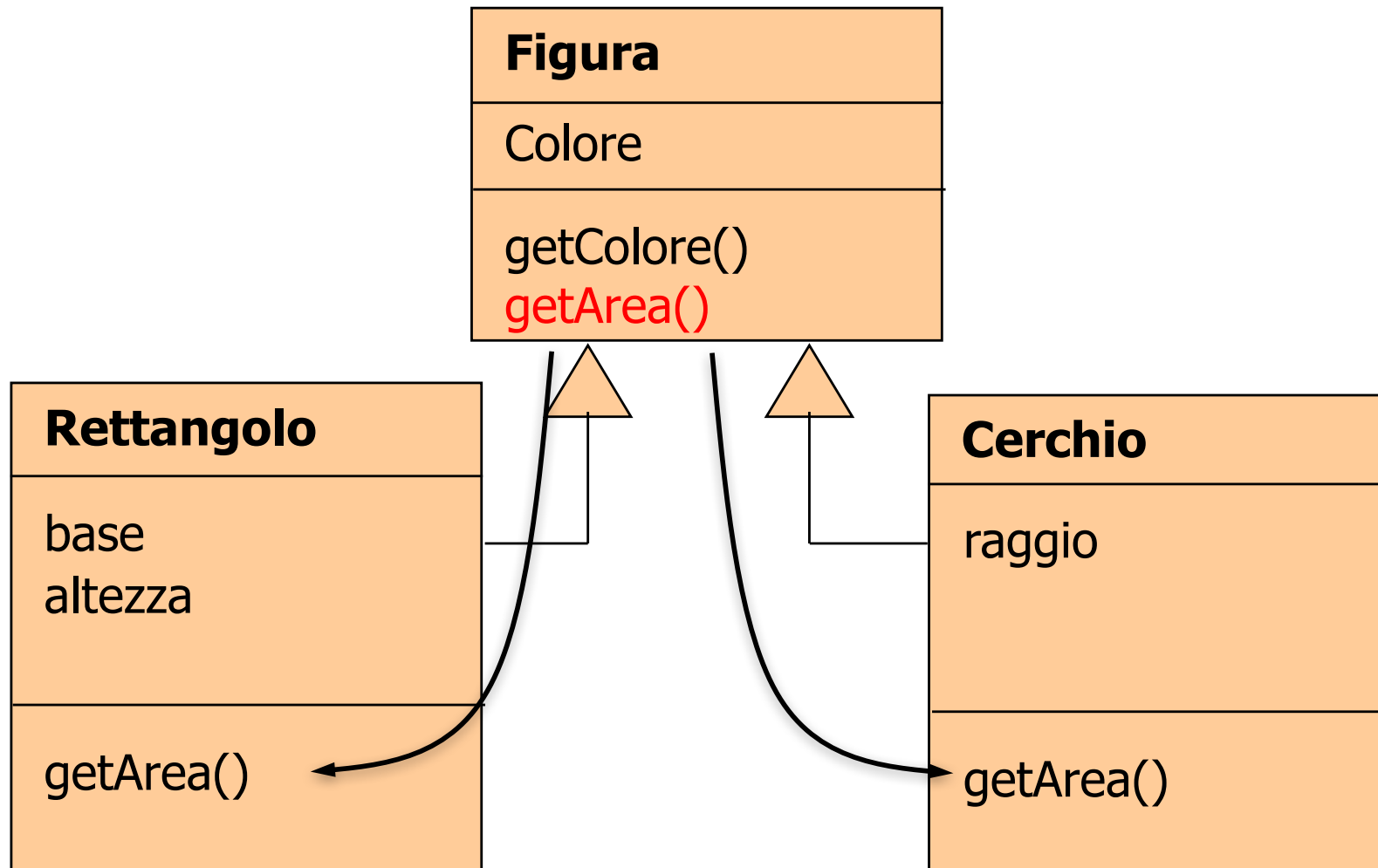




Il problema dell'area

- Una figura ha sempre un'area, che non può essere calcolata a priori
- Un rettangolo ha un suo modo peculiare per calcolare l'area
- Un cerchio ha un altro modo per calcolare l'area

Ereditarietà e binding dinamico





Esempio (3)

```
class Figura{
    private String colore;
    Figura(String col) {
        colore = col;
    }
    String mioColore() {
        return colore;
    }
    double getArea() {
        return 0;
    }
}

class Rettangolo extends Figura {
    private double base, altezza;
    Rettangolo (String col, double alt,
                double bas) {

        super(col);
        altezza = alt;
        base = bas;
    }
    double getArea() {
        return altezza * base;
    }
}
```

```
class Cerchio extends Figura {
    private double raggio;
    Cerchio (String col, double rag) {
        super(col);
        raggio = rag;
    }
    double getArea() {
        return raggio * raggio * 3.14;
    }
}
```



Metodi Astratti

- In *Figura* è presente un metodo *getArea*, impossibile da concretizzare ignorando il tipo di figura
- Si realizza il metodo dichiarandolo ***abstract***
- Tutte le sottoclassi devono fornire un'implementazione del metodo per non essere a loro volta astratte

```
public abstract class Figura {  
    private String colore;  
    public Figura(String col) {  
        colore = col;  
    }  
    String getColore() {  
        return colore;  
    }  
    public abstract double getArea();  
}  
  
class Rettangolo extends Figura {  
    private double altezza;  
    private double base;  
    Rettangolo (String col, double alt, double bas)  
    {  
        super(col);  
        altezza = alt;  
        base = bas;  
    }  
    double getArea() {  
        return altezza * base;  
    }  
}
```



Classi astratte

- Un ibrido tra classe ed interfaccia
 - Ha alcuni metodi normalmente implementati ed altri *astratti*
 - Un metodo astratto non ha implementazione
- ```
public abstract void deductFees() ;
```
- Le classi che estendono una classe astratta sono **OBBLIGATE** ad implementarne i metodi astratti
    - nelle sottoclassi in generale non si è obbligati ad implementare i metodi della superclasse



# Classi astratte

---

- Attenzione: non si possono creare oggetti di classi astratte
  - ...ci sono metodi non implementati (come nelle interfacce!)

```
public abstract class BankAccount {
 public abstract void deductFees();
 ...
}
```



# Classi astratte

---

- E' possibile dichiarare astratta una classe priva di metodi astratti
  - In tal modo evitiamo che possano essere costruiti oggetti di quella classe
- In generale, sono astratte le classi di cui non si possono creare esemplari
- Le classi non astratte sono dette concrete
- Le classi astratte forzano la realizzazione di sottoclassi
- Un metodo astratto consente di non scrivere un metodo fittizio che viene poi ereditato dalle sottoclassi



# Confronto Classi Astratte - Interfacce

---

- Un'interfaccia indica solo dei metodi da implementare
  - consente l'uso di "altre classi" per l'elaborazione di dati
  - può facilmente essere integrata in un progetto sviluppato indipendentemente
  - è consentito implementare più interfacce con la stessa classe
- Una classe astratta fornisce più struttura
  - definisce alcune implementazioni di default
  - permette di definire delle variabili di istanza/statiche/final
  - una classe astratta fornisce una base per le classi che la estenderanno (una classe può estendere una sola superclasse)
- Non è errato usare entrambe in un progetto:
  - l'interfaccia definisce un supertipo per l'utilizzo di un codice (es. DataSet)
  - ciascuna classe astratta è usata per fornire una base alle implementazioni dell'interfaccia



# Metodi e classi final

---

- Per impedire al programmatore di creare sottoclassi o di sovrascrivere certi metodi, si usa la parola chiave **final**
  - **public final class String**
    - questa classe non si può estendere
  - **public final boolean checkPassword(...)**
    - questo metodo non si può sovrascrivere





# Accesso protetto: variabili d'istanza

---

- Nell'implementazione del metodo `deposit` in `CheckingAccount` dobbiamo accedere alla variabile `balance` della superclasse
- Possiamo dichiarare la variabile `balance` protetta

```
public class BankAccount{
 ...
 protected double balance;
}
```
- Ai dati `protected` di un oggetto si può accedere dai metodi della classe, di tutte le sottoclassi, e da tutte le classi che si trovano nello stesso package:
  - `CheckingAccount` è sottoclasse di `BankAccount` e può accedere a `balance`
  - Problema: la sottoclasse può avere metodi aggiuntivi che alterano i dati della superclasse



## Accesso protetto: metodi

---

- **protected** può essere usato per forzare l'uso di alcuni metodi solo da oggetti della stessa classe o di una sottoclasse
  - Si usa in genere per metodi il cui uso corretto dipende dalla conoscenza di dettagli di implementazione
- Un esempio è dato dal metodo **clone** di **Object** (che vedremo in seguito)



# Ereditarietà e specificatori di accesso

---

- Quando si sovrascrivono i metodi di una superclasse non se ne può restringere la visibilità
  - Ad esempio: un metodo dichiarato `protected` può essere sovrascritto in una sottoclasse assegnando specificatore d'accesso `protected` o `public` ma non `package` o `private`.



# Controllo di accesso a variabili, metodi e classi (specificatori di accesso)

---

| <b>Accessibile da</b>                                       | <b>public</b> | <b>package</b> | <b>private</b> | <b>protected</b> |
|-------------------------------------------------------------|---------------|----------------|----------------|------------------|
| <b>Stessa Classe</b>                                        | Si            | Si             | Si             | Si               |
| <b>Altra Classe<br/>(stesso package)</b>                    | Si            | Si             | No             | Si               |
| <b>Altra Classe non<br/>sottoclasse<br/>(altro package)</b> | Si            | No             | No             | No               |
| <b>Sottoclasse (altro<br/>package)</b>                      | Si            | No             | No             | Si               |



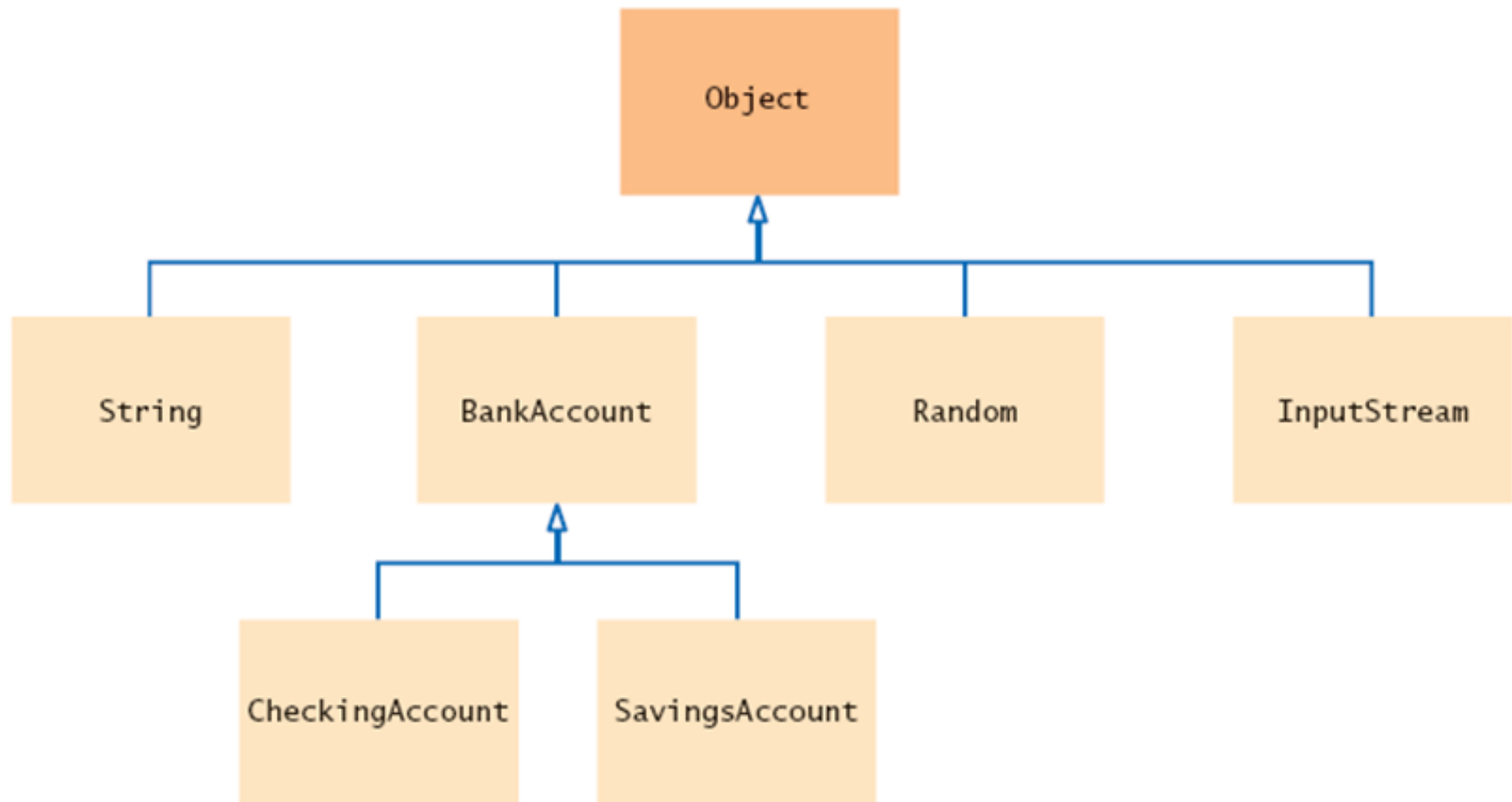
# Object: La classe universale

---

- Ogni classe che non estende un'altra classe, estende per default la classe **Object**
- Metodi della classe **Object**
  - **String toString()**
    - Restituisce una rappresentazione dell'oggetto in forma di stringa
  - **boolean equals(Object otherObject)**
    - Verifica se l'oggetto è uguale a un altro
  - **Object clone()**
    - Crea una copia dell'oggetto
- E' opportuno sovrascrivere questi metodi nelle nostre classi

# Object: La classe universale

---



La classe **Object** è la superclasse di tutte le classi Java



# Sovrascrivere toString

---

- Restituisce una stringa contenente lo stato dell'oggetto.  

```
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
String s = cerealBox.toString();
// s si riferisce alla stringa
// "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```
- Automaticamente invocato quando si concatena una stringa con un oggetto:  

```
"cerealBox=" +cerealBox
```

viene valutata:

```
"cerealBox =
java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```



## Sovrascrivere `toString`

---

- L'operazione vista prima funziona solo se uno dei due oggetti è già una stringa
  - Il compilatore può invocare `toString()` su qualsiasi oggetto, dato che ogni classe estende la classe `Object`
- Se nessuno dei due oggetti è una stringa il compilatore genera un errore





# Sovrascrivere toString

---

- Proviamo a usare il metodo `toString()` nella classe `BankAccount`:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
//s si riferisce a "BankAccount@d24606bf"
```

- Viene stampato il nome della classe seguito dall'indirizzo in memoria dell'oggetto (codice hash)
- Ma noi volevamo sapere cosa si trova nell'oggetto!
  - Il metodo `toString()` della classe `Object` non può sapere cosa si trova all'interno della classe `BankAccount`



# Sovrascrivere toString

---

- Dobbiamo sovrascrivere il metodo nella classe **BankAccount**:

```
public String toString()
{
 return "BankAccount[balance=" + balance + "];"
}
```

- In tal modo:

```
BankAccount momsSavings = new
 BankAccount(5000);
String s = momsSavings.toString();
//s si riferisce a "BankAccount[balance=5000]"
```



# Sovrascrivere `toString`

---

- E' importante fornire il metodo `toString()` in tutte le classi!
  - Ci consente di controllare lo stato di un oggetto
  - Se `x` è un oggetto e abbiamo sovrascritto `toString()`, possiamo invocare `System.out.println(x)`
    - Il metodo `println` della classe `PrintStream` invoca `x.toString()`



# Sovrascrivere toString

---

- E' preferibile non inserire il nome della classe, ma `getClass().getName()`
  - Il metodo `getClass()`
    - consente di sapere il tipo esatto dell'oggetto a cui punta un riferimento.
    - metodo della classe `Object`
- Restituisce un oggetto di tipo `Class`, da cui possiamo ottenere informazioni relative alla classe
  - `Class c = e.getClass()`
- Ad esempio, il metodo `getName()` della classe `Class` restituisce la stringa contenente il nome della classe

```
public String toString()
{
 return getClass().getName() + "[balance=" + balance + "];"
}
```



# Sovrascrivere toString

---

- Ora possiamo invocare `toString()` anche su un oggetto della sottoclasse

```
SavingsAccount sa = new SavingsAccount(10);
System.out.println(sa);
// stampa "SavingsAccount[balance=1000]";
// non stampa anche il contenuto di
// interestRate!
```



## Sottoclassi: sovrascrivere toString

---

- Nella sottoclasse dobbiamo sovrascrivere **toString()** e aggiungere i valori delle variabili istanza della sottoclasse

```
public class SavingsAccount extends BankAccount
{
 public String toString()
 {
 return super.toString() + "[interestRate="
 + interestRate + "];"
 }
}
```



## Sottoclassi: sovrascrivere toString

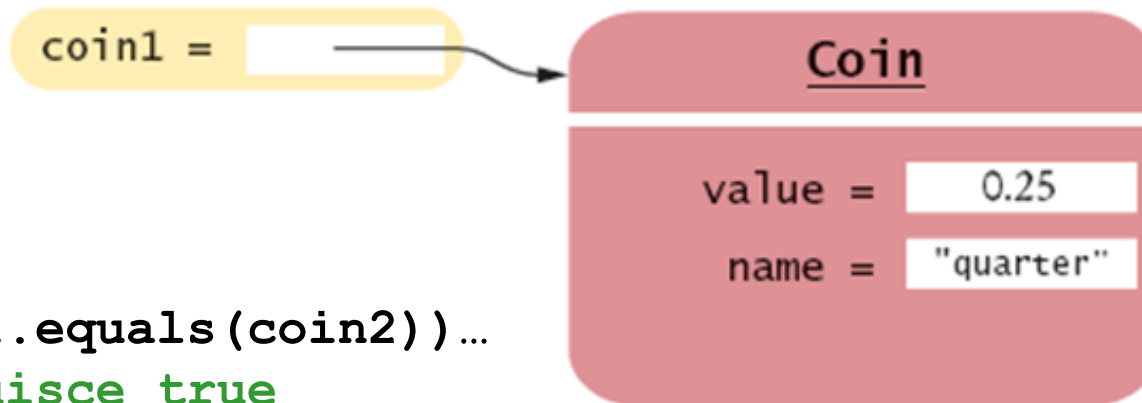
---

- Vediamo la chiamata su un oggetto di tipo **SavingsAccount**:

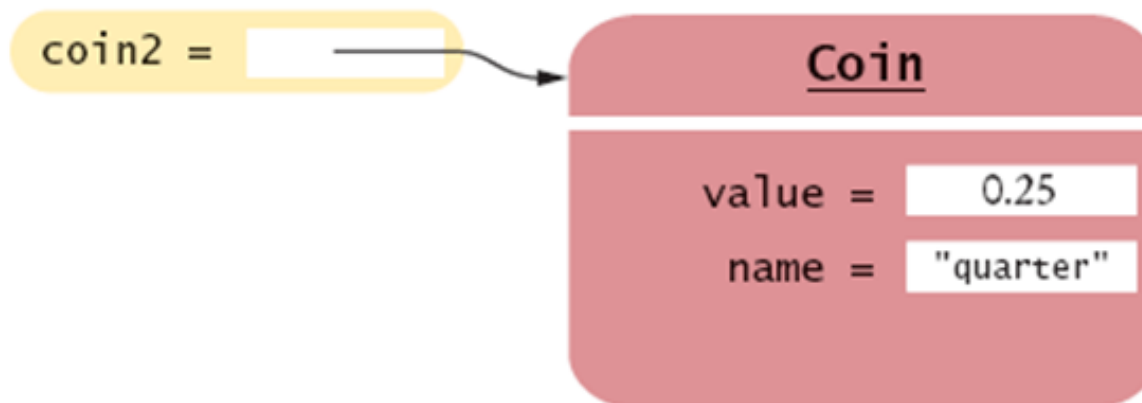
```
SavingsAccount sa = new SavingsAccount(10);
System.out.println(sa);
//stampa "SavingsAccount[balance=1000]
 [interestRate=10]";
```

# Sovrascrivere `equals`

- Il metodo `equals` verifica se due oggetti hanno lo stesso contenuto



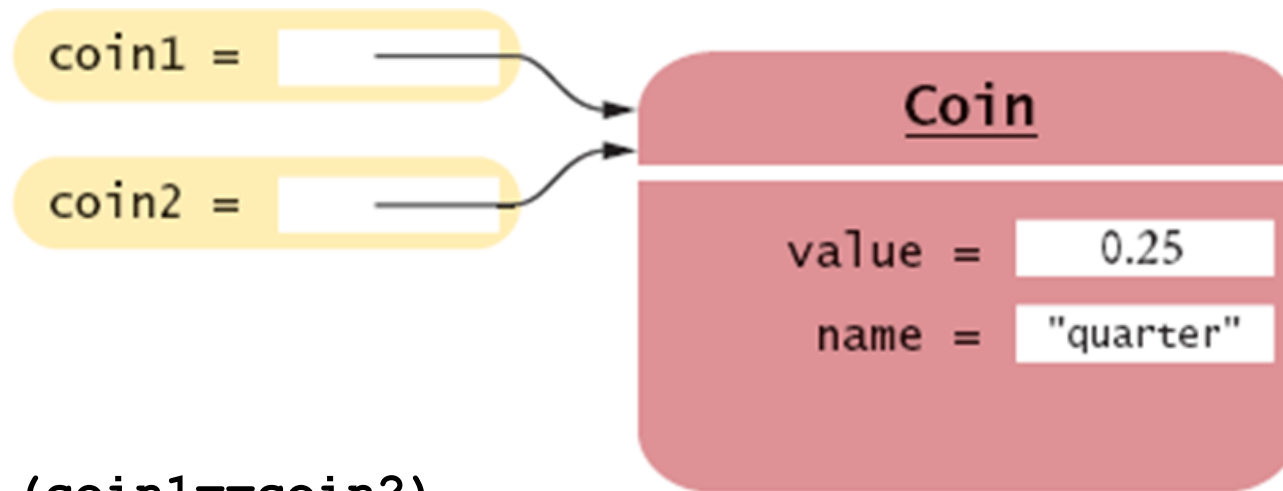
```
if (coin1.equals(coin2))...
//restituisce true
```





# Sovrascrivere `equals`

- L'operatore `==` verifica se due riferimenti indicano lo stesso oggetto



```
if (coin1==coin2)...
//restituisce true
```



# Sovrascrivere `equals`

---

```
boolean equals(Object otherObject) {
}
```

- Sovrascriviamo il metodo `equals` nella classe **Coin**
  - Problema: il parametro `otherObject` è di tipo **Object** e non **Coin**
  - Se riscriviamo il metodo non possiamo variare la firma, ma dobbiamo eseguire un cast sul parametro  
`Coin other = (Coin)otherObject;`



# Sovrascrivere `equals`

---

- Ora possiamo confrontare le monete:

```
public boolean equals(Object otherObject) {
 Coin other = (Coin) otherObject;
 return name.equals(other.name)
 && value == other.value;
}
```

- Controlla se hanno lo stesso nome e lo stesso valore
  - Per confrontare `name` e `other.name` usiamo `equals` perché si tratta di riferimenti a stringhe
  - Per confrontare `value` e `other.value` usiamo `==` perché si tratta di variabili numeriche



# Sovrascrivere equals

---

- Se invochiamo `coin1.equals(x)` e `x` non è di tipo `Coin`?
  - Il cast errato eseguito in seguito genera un'eccezione
- Possiamo usare `instanceof` per controllare se `x` è di tipo `Coin`

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



# Sovrascrivere `equals`

---

- Se uso `instanceof` per controllare se una classe è di un certo tipo, la risposta sarà `true` anche se l'oggetto appartiene a qualche sottoclasse...
- Dovrei verificare se i due oggetti appartengano alla stessa classe:

```
if (getClass() != otherObject.getClass())
 return false;
```
- Infine, `equals` dovrebbe restituire `false` se `otherObject` è `null`



## 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`

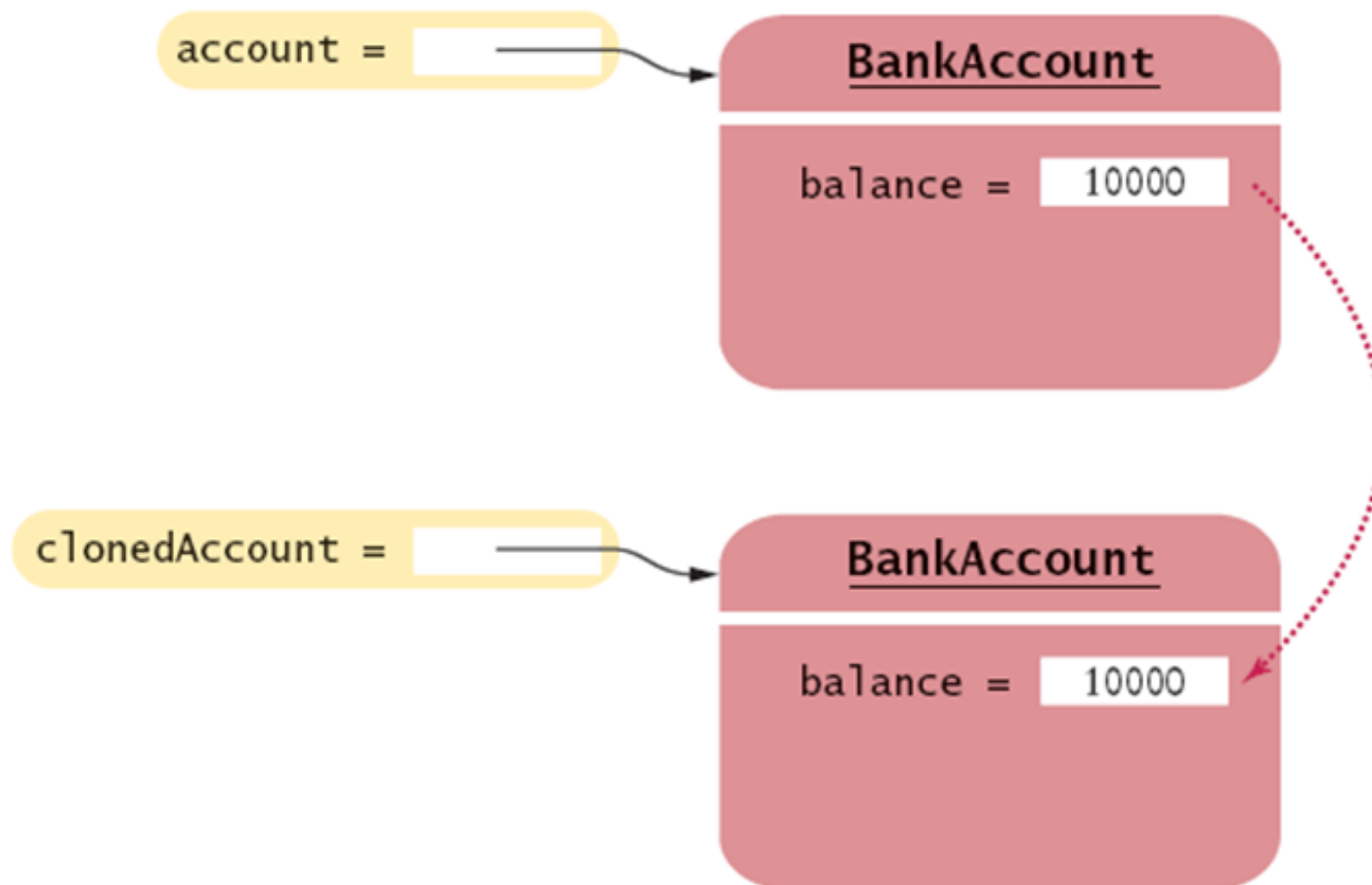
---

- 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`

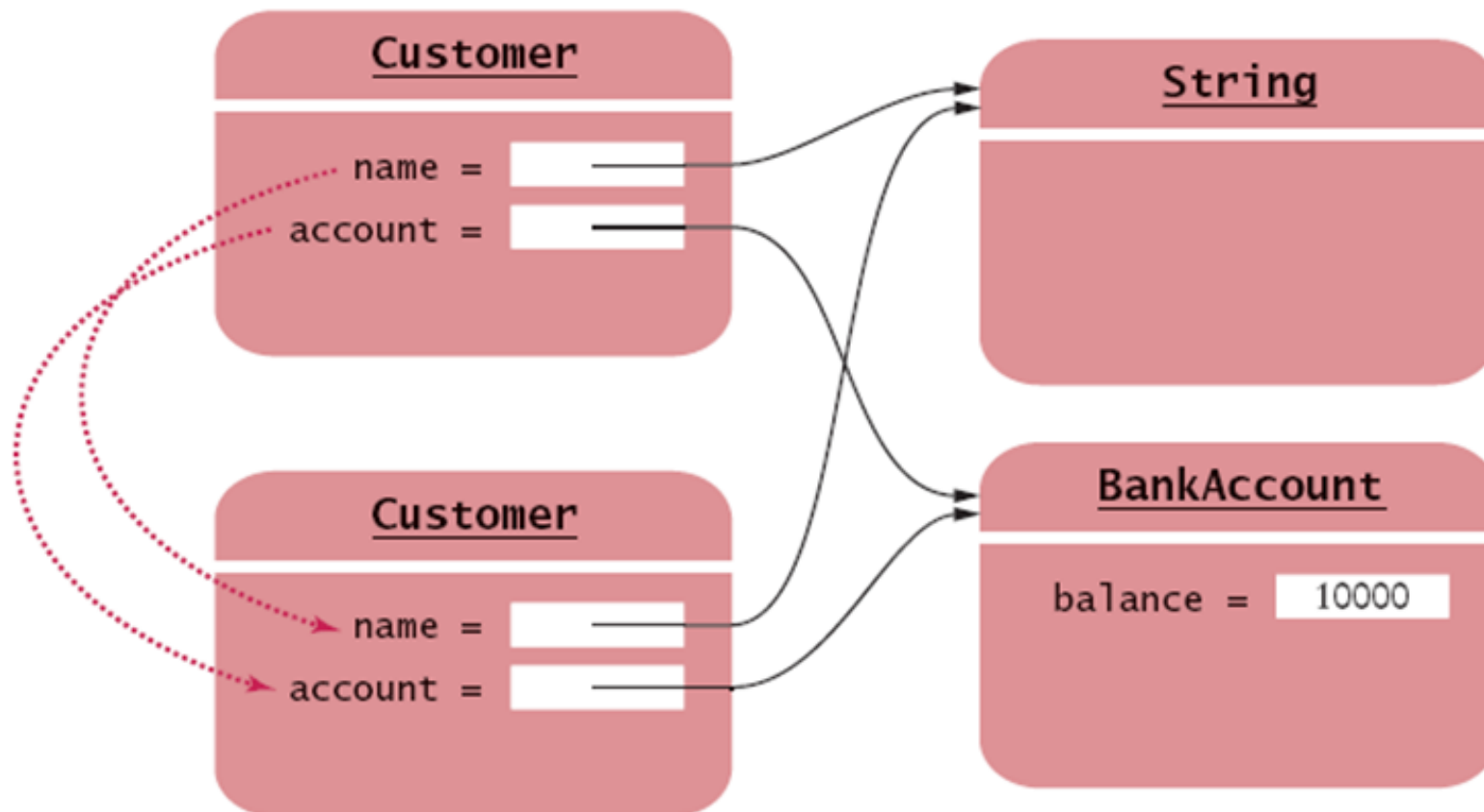
---

- 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

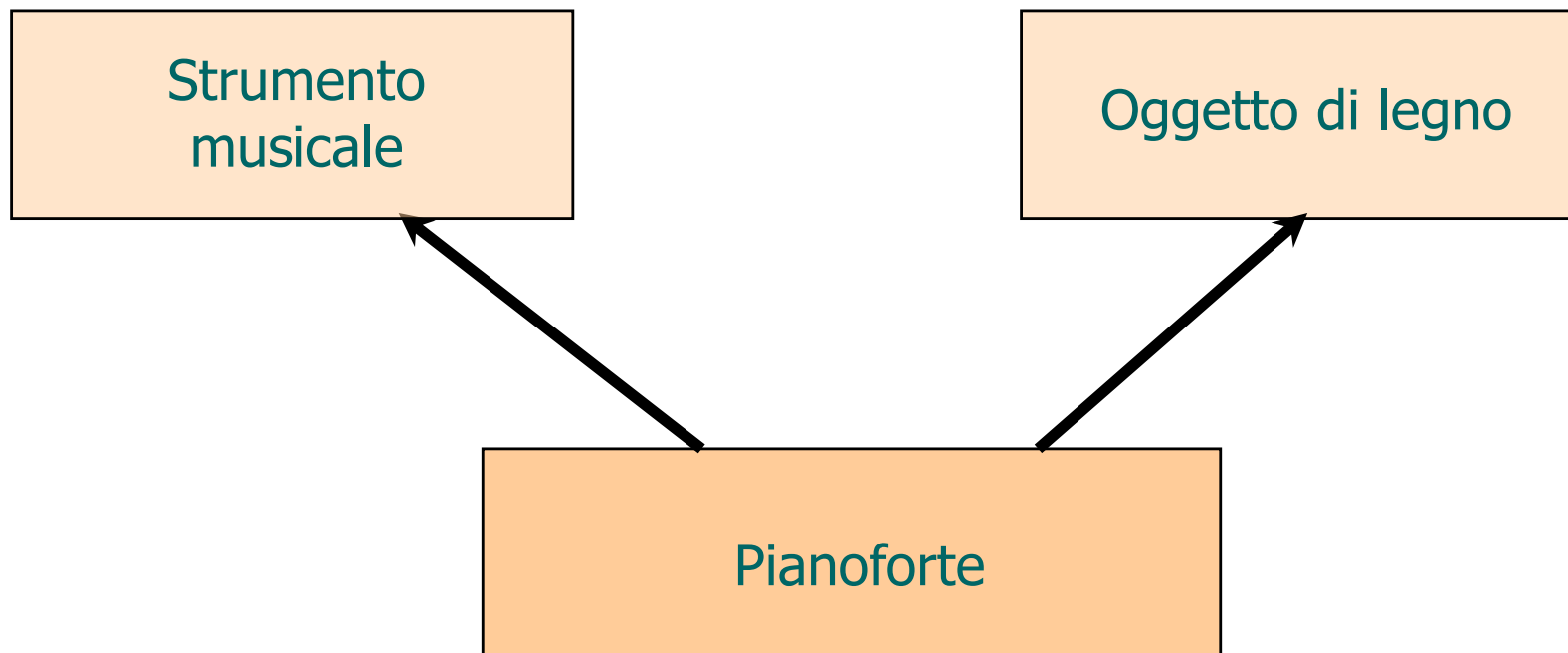
---

```
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*

---

