



# Ereditarietà

---



# Ereditarietà

---

- Ereditarietà come meccanismo di estensione del comportamento
- Esempi:
  - Aggiungere funzionalità di colori ad una classe Finestra
  - Aggiungere capacità di ordinamento ad una classe Agenda
  - Aggiungere un *middle name* alla classe Name
- Modifichiamo la classe esistente ?
- **Potrebbe non essere desiderabile**
  - La classe è già rigorosamente verificata e robusta
  - Allargare la classe comporta una complessità aggiuntiva
  - Il codice sorgente potrebbe non essere disponibile
  - Le modifiche possono non essere consigliabili (ad esempio per le classi Java predefinite)



# Ereditarietà

---

- E' un meccanismo per estendere classi esistenti, aggiungendo altri metodi e campi.

```
public class SavingsAccount extends BankAccount
{
    nuovi metodi
    nuove variabili d'istanza
}
```

- Tutti i metodi e le variabili d'istanza della classe **BankAccount** sono ereditati automaticamente
- Consente il riutilizzo del codice



# Ereditarietà

---

- La classe preesistente (più generica) è detta **SUPERCLASSE** e la nuova classe (più specifica) è detta **SOTTOCLASSE**
  - **BankAccount**: superclasse
  - **SavingsAccount**: sottoclasse



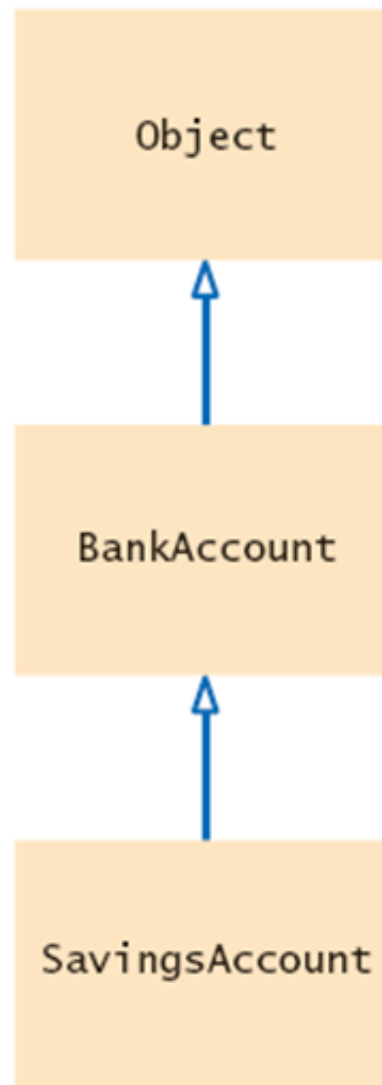
# Base comune a tutte le classi

---

- La classe **Object** è la superclasse di tutte le classi.
  - Ogni classe è una sottoclasse di **Object**
- Ha un piccolo numero di metodi, tra cui
  - **String toString()**
  - **boolean equals(Object otherObject)**
  - **Object clone()**

# Diagramma di ereditarietà

---





# Ereditarietà vs Interfacce

---

- Differenza con l'implementazione di una interfaccia:
  - un'interfaccia non è una classe
    - non ha uno stato, né un comportamento
    - è un elenco di metodi da implementare
  - una sottoclasse è una classe
    - ha uno stato e un comportamento che sono ereditati dalla superclasse



# Riutilizzo di codice

---

- La classe **SavingsAccount** eredita i metodi della classe **BankAccount**:
  - **withdraw**
  - **deposit**
  - **getBalance**
- Inoltre, **SavingsAccount** ha un metodo che calcola gli interessi maturati e li versa sul conto
  - **addInterest**





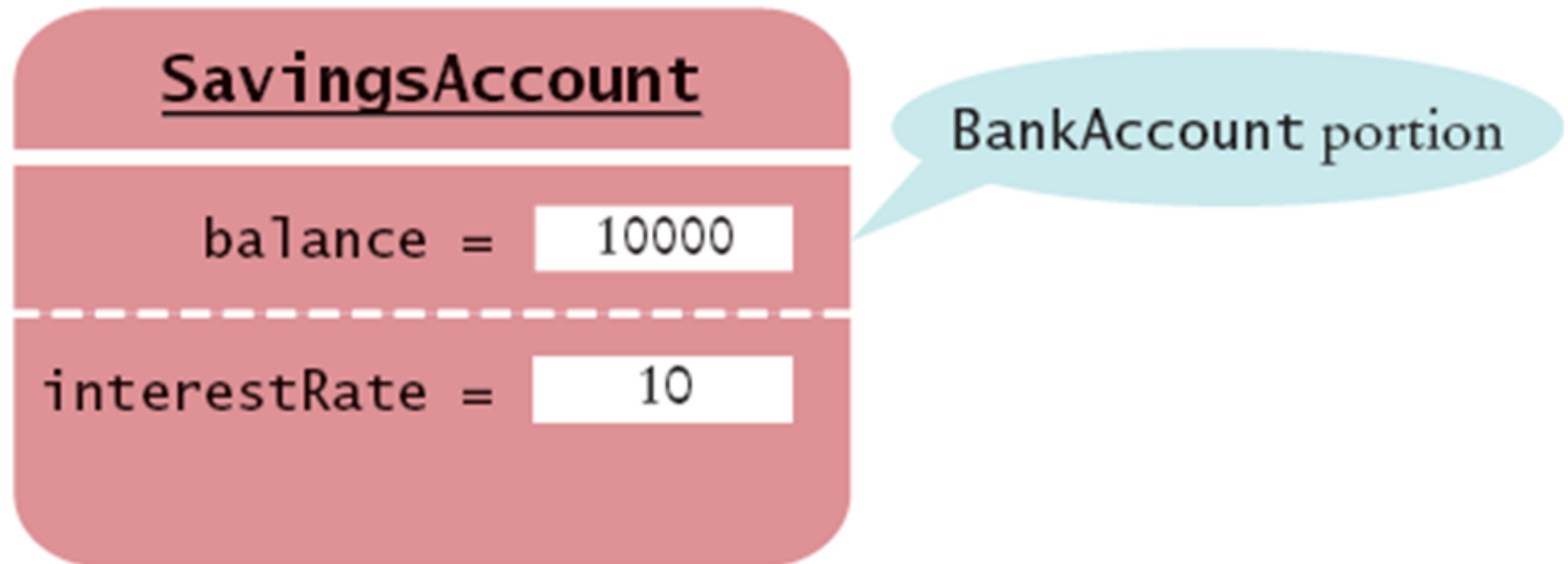
# Classe: SavingsAccount

---

```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount(double rate)
    {
        interestRate = rate;
    }
    public void addInterest()
    {
        double interest = getBalance()
                           * interestRate / 100;
        deposit(interest);
    }
    private double interestRate;
}
```

# Classe: SavingsAccount

---



**SavingsAccount** eredita la variabile di istanza `balance` da **BankAccount** e ha una variabile di istanza in più: `interestRate`



# Classe: SavingsAccount

---

- Il metodo **addInterest** chiama i metodi **getBalance** e **deposit** della superclasse
  - Non viene specificato alcun oggetto per le invocazioni di tali metodi
  - Viene usato il parametro implicito di **addInterest**

```
double interest = this.getBalance()  
                * this.interestRate / 100;  
  
this.deposit(interest);
```
  - Non si può usare direttamente **balance**
    - è dichiarato **private** in **BankAccount**



# Classe: SavingsAccount

---

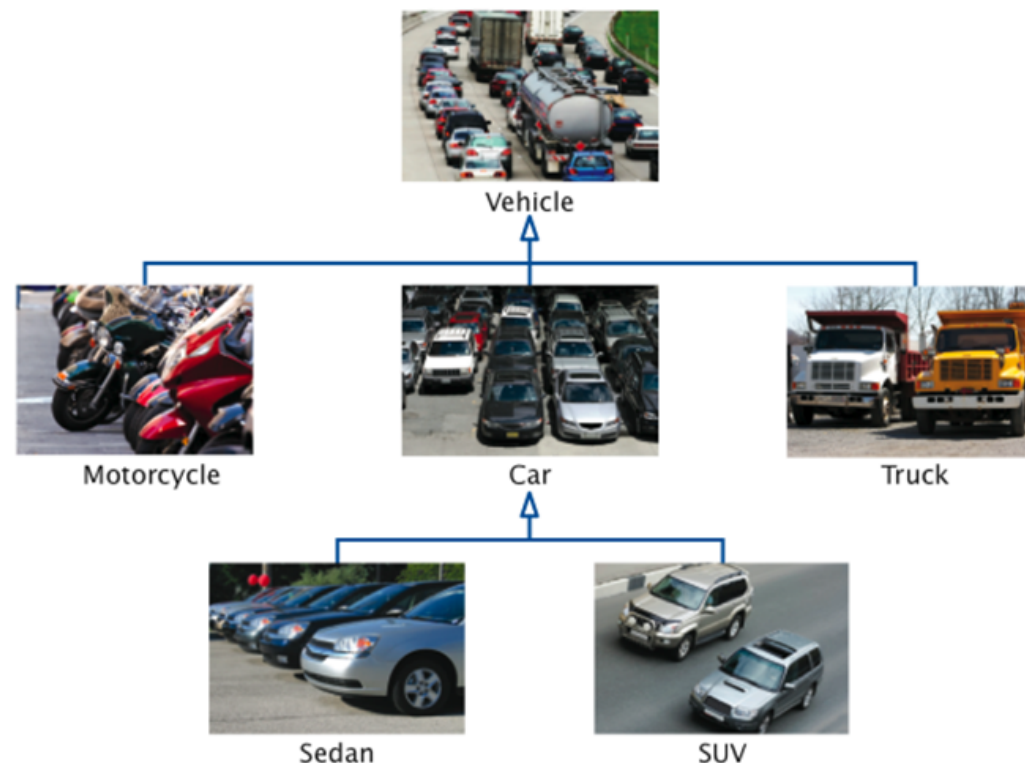
- `SavingsAccount sa =  
new SavingsAccount(10);`

- `sa.addInterest();`
  - Viene usato il parametro implicito di `addInterest`

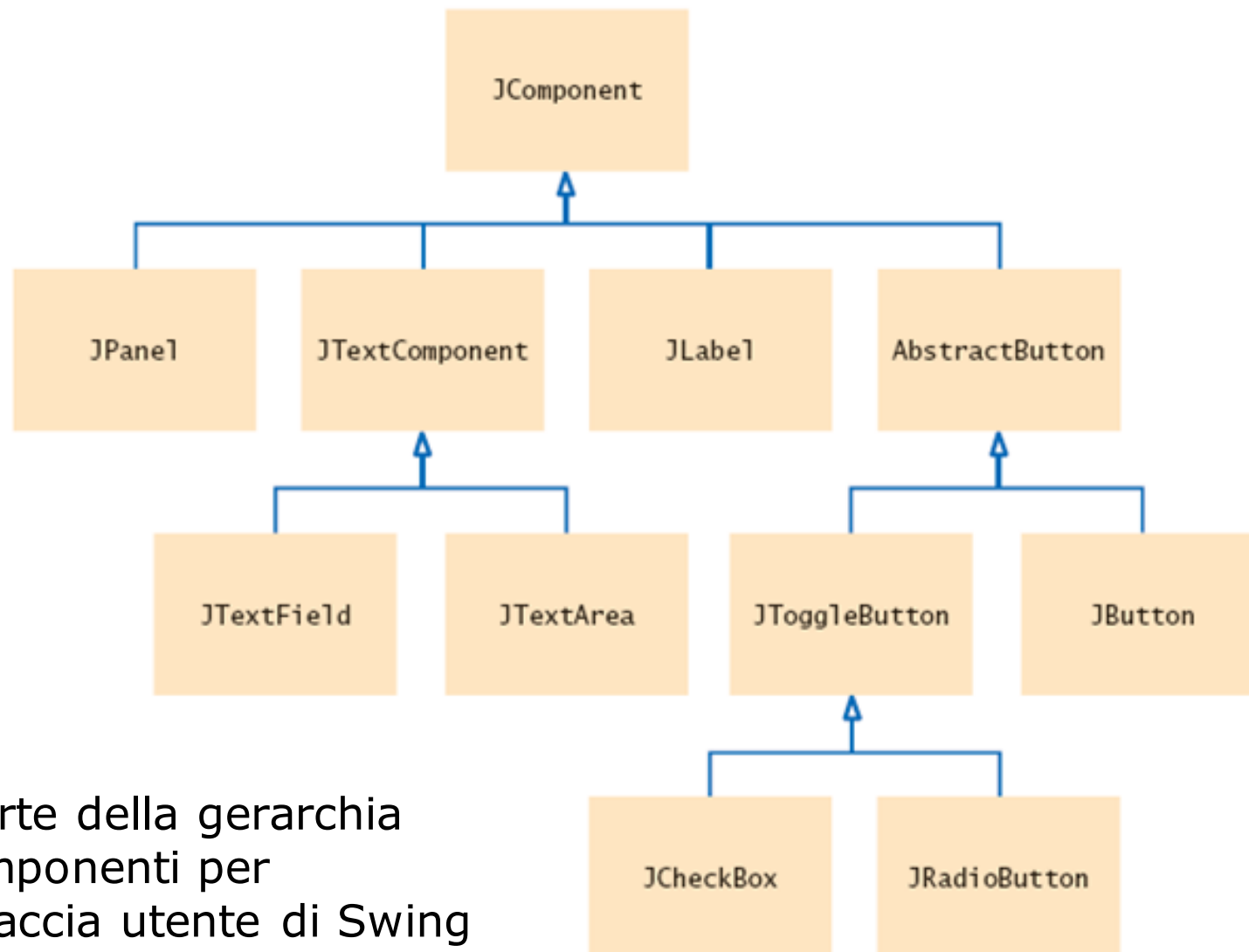
```
double interest = sa.getBalance()  
                * sa.interestRate / 100;  
sa.deposit(interest);
```

# Gerarchie di ereditarietà

- In Java le classi sono raggruppate in gerarchie di ereditarietà
  - Le classi che rappresentano concetti più generali sono più vicine alla radice
  - Le classi più specializzate sono nelle diramazioni



# Gerarchie di ereditarietà

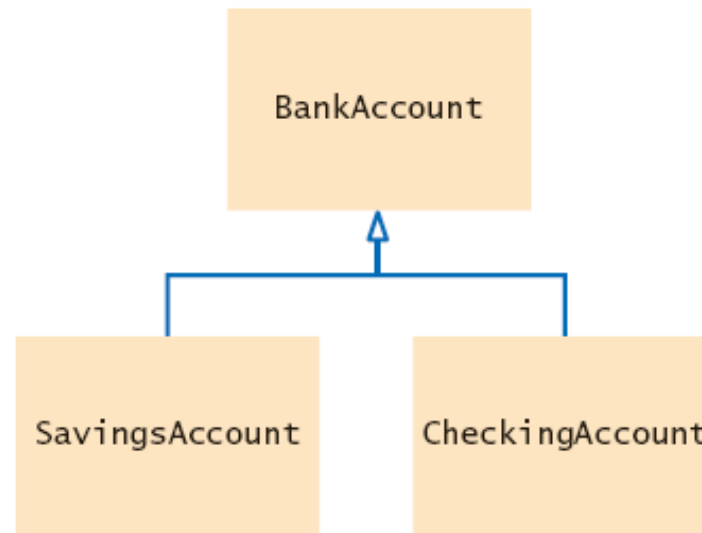


Una parte della gerarchia  
dei componenti per  
l'interfaccia utente di Swing

# Gerarchie di ereditarietà

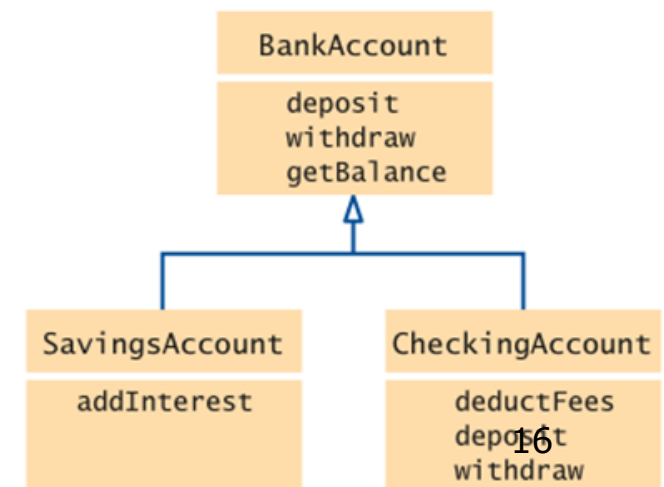
---

- Consideriamo una banca che offre due tipi di conto:
  - **Checking account**, che non offre interessi, concede un certo numero di operazioni mensili gratuite e addebita una commissione per ogni operazione aggiuntiva
  - **Savings account**, che frutta interessi mensili



# Gerarchie di ereditarietà

- Determiniamo i comportamenti:
  - Tutti i conti forniscono i metodi
    - **getBalance**, **deposit** e **withdraw**
  - Per **CheckingAccount** bisogna contare le transazioni
  - Per **CheckingAccount** è necessario un metodo per addebitare le commissioni mensili
    - **deductFees**
  - **SavingsAccount** ha un metodo per sommare gli interessi
    - **addInterest**







# Metodi di una sottoclasse

---

Tre possibilità per definirli:

- Sovrascrivere metodi della superclasse
  - la sottoclasse ridefinisce un metodo con la stessa firma del metodo della superclasse
  - vale il metodo della sottoclasse
- Ereditare metodi dalla superclasse
  - la sottoclasse non ridefinisce nessun metodo della superclasse
- Definire nuovi metodi
  - la sottoclasse definisce un metodo che non esiste nella superclasse



# Variabili di istanza di sottoclassi

---

Due possibilità:

- Ereditare variabili istanza
  - Le sottoclassi ereditano tutte le variabili di istanza della superclasse
- Definire nuove variabili istanza
  - Esistono solo negli oggetti della sottoclasse
  - Possono avere lo stesso nome di quelle nella superclasse, ma non sono sovrascritte
  - Quelle della sottoclasse mettono in ombra quelle della superclasse



# La nuova classe: CheckingAccount

---

```
public class BankAccount {  
    public double getBalance() {...}  
    public void deposit(double d) {...}  
    public void withdraw(double d) {...}  
    private double balance;  
}
```

```
public class CheckingAccount extends BankAccount {  
    public void deposit(double d) {...}  
    public void withdraw(double d) {...}  
    public void deductFees() {...}  
    private int transactionCount;  
}
```



# CheckingAccount

---

- Ciascun oggetto di tipo **CheckingAccount** ha due variabili di istanza
  - **balance** (ereditata da **BankAccount**)
  - **transactionCount** (nuova)
- E' possibile applicare quattro metodi
  - **getBalance** () (ereditato da **BankAccount**)
  - **deposit** (**double**) (sovrascritto)
  - **withdraw** (**double**) (sovrascritto)
  - **deductFees** () (nuovo)



# CheckingAccount: metodo deposit

---

```
public void deposit(double amount)
{
    transactionCount++; // NUOVA VBL IST.

    //aggiungi amount al saldo
    balance = balance + amount; //ERRORE
}
```

- **CheckingAccount** ha una variabile **balance**, ma è una variabile privata della superclasse!
- I metodi della sottoclasse non possono accedere alle variabili private della superclasse



# CheckingAccount: metodo deposit

---

- Possiamo invocare il metodo `deposit` della classe `BankAccount`...

- Ma se scriviamo

`deposit(amount)`

viene interpretato come

`this.deposit(amount)`

cioè viene chiamato il metodo che stiamo scrivendo!

- Dobbiamo chiamare il metodo `deposit` della superclasse:

`super.deposit(amount)`



# CheckingAccount: metodo deposit

---

```
public void deposit(double amount)
{
    transactionCount++; // NUOVA VBL IST.

    //aggiungi amount al saldo
    super.deposit(amount) ;
}
```



# CheckingAccount: metodo `withdraw`

---

```
public void withdraw(double amount)
{
    transactionCount++; // NUOVA VBL IST.

    //sottrai amount al saldo
    super.withdraw(amount);
}
```





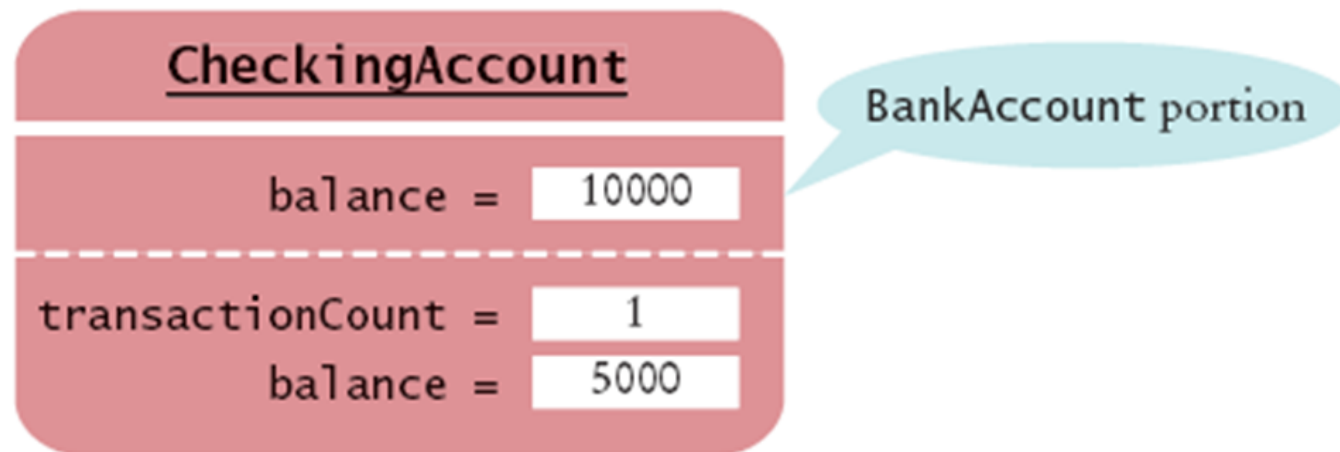
# CheckingAccount: metodo deductFees

---

```
public void deductFees ()
{
    if (transactionCount > FREE_TRANSACTIONS) {
        double fees = TRANSACTION_FEE*
            (transactionCount - FREE_TRANSACTIONS) ;
        super.withdraw(fees) ;
    }
    transactionCount = 0 ;
}
```

# Mettere in ombra variabili istanza

- Una sottoclasse non ha accesso alle variabili private della superclasse
- E' un errore comune risolvere il problema creando un'altra variabile di istanza con lo stesso nome
- La variabile della sottoclasse mette in ombra quella della superclasse





# Costruzione di sottoclassi

---

- Per invocare il costruttore della superclasse dal costruttore di una sottoclasse uso la parola chiave **super** seguita dai parametri del costruttore
  - Deve essere il primo comando del costruttore della sottoclasse

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount(double initialBalance)
    {
        super(initialBalance);
        transactionCount = 0;
    }
}
```



# Costruzione di sottoclassi

---

- Se il costruttore della sottoclasse non chiama il costruttore della superclasse, viene invocato il costruttore predefinito della superclasse
  - Se il costruttore di **CheckingAccount** non invoca il costruttore di **BankAccount**, viene impostato il saldo iniziale a zero

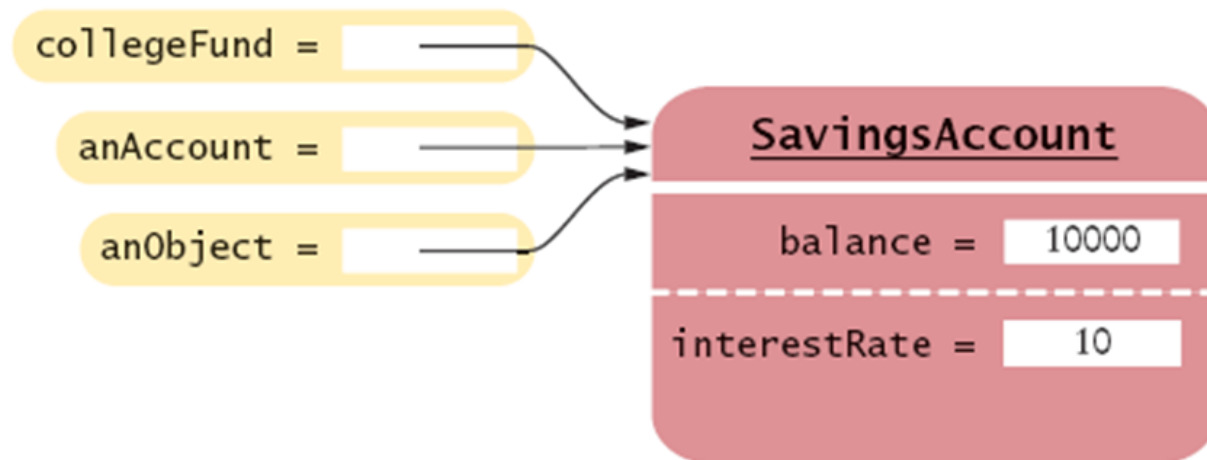
# Conversione da Sottoclasse a Superclasse

- Si può salvare un riferimento ad una sottoclasse in una variabile di riferimento ad una superclasse:

```
SavingsAccount collegeFund = new SavingsAccount(10);  
BankAccount anAccount = collegeFund;
```

- Il riferimento a qualsiasi oggetto può essere memorizzato in una variabile di tipo **Object**

```
Object anObject = collegeFund;
```





# Conversione da Sottoclasse a Superclasse

---

- Non si possono applicare metodi della sottoclasse:

```
anAccount.deposit(1000); //Va bene  
//deposit è un metodo della classe BankAccount
```

```
anAccount.addInterest(); // Errore  
//addInterest non è un metodo della classe  
BankAccount
```

```
anObject.deposit(); // Errore  
//deposit non è un metodo della classe Object
```



# Polimorfismo

---

- Vi ricordate il metodo **transfer**:

```
public void transfer(BankAccount other, double amount)
{
    withdraw(amount);
    other.deposit(amount);
}
```

- Gli si può passare qualsiasi tipo di **BankAccount**



# Polimorfismo

---

- E' lecito passare un riferimento di tipo **CheckingAccount** a un metodo che si aspetta un riferimento di tipo **BankAccount**

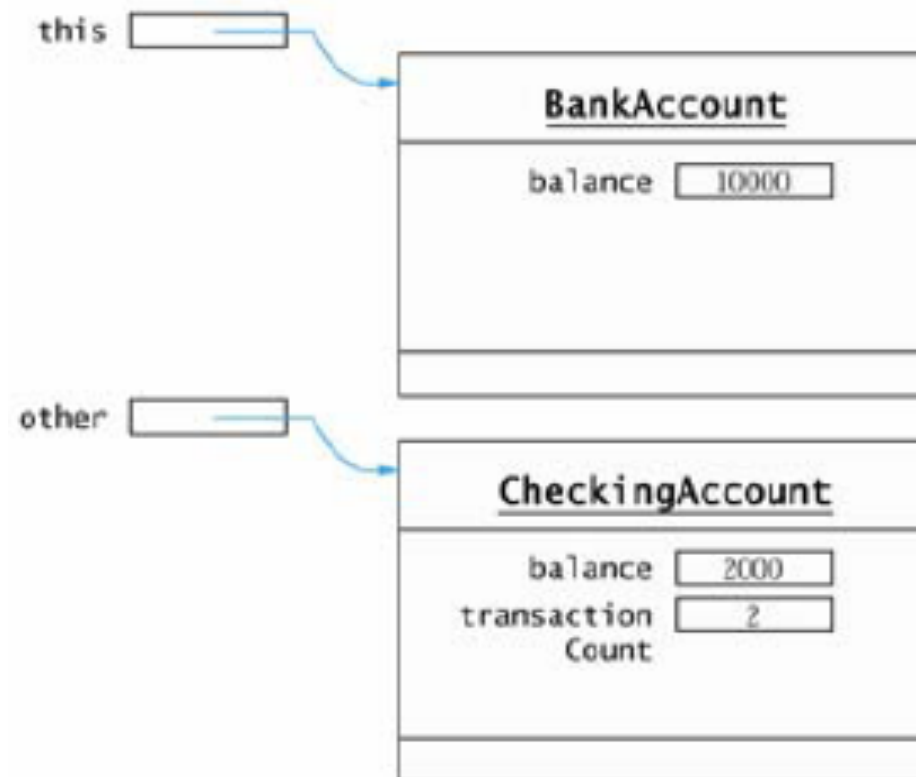
```
BankAccount momsAccount = . . . ;  
CheckingAccount harrysChecking = . . . ;  
momsAccount.transfer(harrysChecking, 1000) ;
```

- Il compilatore copia il riferimento all'oggetto **harrisChecking** di tipo sottoclasse nel riferimento di superclasse **other**



# Polimorfismo

- Il metodo transfer non sa che **other** si riferisce a un oggetto di tipo **CheckingAccount**
- Sa solo che **other** è un riferimento di tipo **BankAccount**





# Polimorfismo

---

- Il metodo `transfer` invoca il metodo `deposit`.
  - Quale metodo?
- Dipende dal tipo reale dell'oggetto (`late binding`)
  - Su un oggetto di tipo `CheckingAccount` viene invocato `CheckingAccount.deposit( )`
- Vediamo un programma che chiama i metodi polimorfici `withdraw` e `deposit`



# File BankAccount.java

---

```
01: /**
02:     A bank account has a balance that can be changed by
03:     deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:     /**
08:         Constructs a bank account with a zero balance.
09:     */
10:     public BankAccount()
11:     {
12:         balance = 0;
13:     }
14:
15:     /**
16:         Constructs a bank account with a given balance.
17:         @param initialBalance the initial balance
18:     */
19:     public BankAccount(double initialBalance)
20:     {
21:         balance = initialBalance;
22:     }
23:
```



# File BankAccount.java

```
24:    /**
25:        Deposits money into the bank account.
26:        @param amount the amount to deposit
27:    */
28:    public void deposit(double amount)
29:    {
30:        balance = balance + amount;
31:    }
32:
33:    /**
34:        Withdraws money from the bank account.
35:        @param amount the amount to withdraw
36:    */
37:    public void withdraw(double amount)
38:    {
39:        balance = balance - amount;
40:    }
41:
42:    /**
43:        Gets the current balance of the bank account.
44:        @return the current balance
45:    */
```



# File BankAccount.java

---

```
46:     public double getBalance()
47:     {
48:         return balance;
49:     }
50:
51:     /**
52:      * Transfers money from the bank account to another account
53:      * @param amount the amount to transfer
54:      * @param other the other account
55:      */
56:     public void transfer(double amount, BankAccount other)
57:     {
58:         withdraw(amount);
59:         other.deposit(amount);
60:     }
61:
62:     private double balance;
63: }
```



# File CheckingAccount.java

---

```
/**
Un conto corrente che addebita commissioni
per ogni transazione.
*/
public class CheckingAccount extends BankAccount
{
    /**
    Costruisce un conto corrente con un saldo assegnato.
    @param initialBalance il saldo iniziale
    */
    public CheckingAccount(double initialBalance)
    {
        // chiama il costruttore della superclasse
        super(initialBalance);
        // inizializza il conteggio delle transazioni
        transactionCount = 0;
    }
}
```



# File CheckingAccount.java

---

```
//metodo sovrascritto
public void deposit(double amount){
    transactionCount++;
    // ora aggiungi amount al saldo
    super.deposit(amount) ;
}

//metodo sovrascritto
public void withdraw(double amount){
    transactionCount++;
    // ora sottrai amount dal saldo
    super.withdraw(amount) ;
}
```



# File CheckingAccount.java

---

```
//metodo nuovo
public void deductFees(){
    if (transactionCount > FREE_TRANSACTIONS){
        double fees = TRANSACTION_FEE *
            (transactionCount - FREE_TRANSACTIONS);
        super.withdraw(fees);
    }
    transactionCount = 0;
}

private int transactionCount;
private static final int FREE_TRANSACTIONS = 3;
private static final double TRANSACTION_FEE = 2.0;
}
```





# File SavingsAccount.java

---

```
/**
    Un conto bancario che matura interessi ad un
    tasso fisso.
*/
public class SavingsAccount extends BankAccount{
    /**
        Costruisce un conto bancario con un tasso di
        interesse assegnato.
        @param rate il tasso di interesse
    */
    public SavingsAccount(double rate){
        interestRate = rate;
    }
}
```



# File SavingsAccount.java

---

```
/**
    Aggiunge al saldo del conto gli interessi
    maturati.
 */
public void addInterest()
{
    double interest = getBalance()
                      * interestRate / 100;
    deposit(interest);
}
private double interestRate;
}
```



# File AccountTest.java

---

```
/**
    Questo programma collauda la classe BankAccount
    e le sue sottoclassi.
 */
public class AccountTest{
    public static void main(String[] args){
        BankAccount momsSavings
            = new SavingsAccount(0.5);

        BankAccount harrysChecking
            = new CheckingAccount(100);
        momsSavings.deposit(10000);
        momsSavings.transfer(2000, harrysChecking);
        harrysChecking.withdraw(1500);
        harrysChecking.withdraw(80);
    }
}
```



# File AccountTest.java

---

```
momsSavings.transfer(1000, harrysChecking);
harrysChecking.withdraw(400);

// simulazione della fine del mese
((SavingsAccount) momsSavings).addInterest();
((CheckingAccount) harrysChecking).deductFees();

System.out.println("Mom's savings balance = $"
                   + momsSavings.getBalance());

System.out.println("Harry's checking balance = $"
                   + harrysChecking.getBalance());
}
}
```



# 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() {...}  
    ...  
}
```





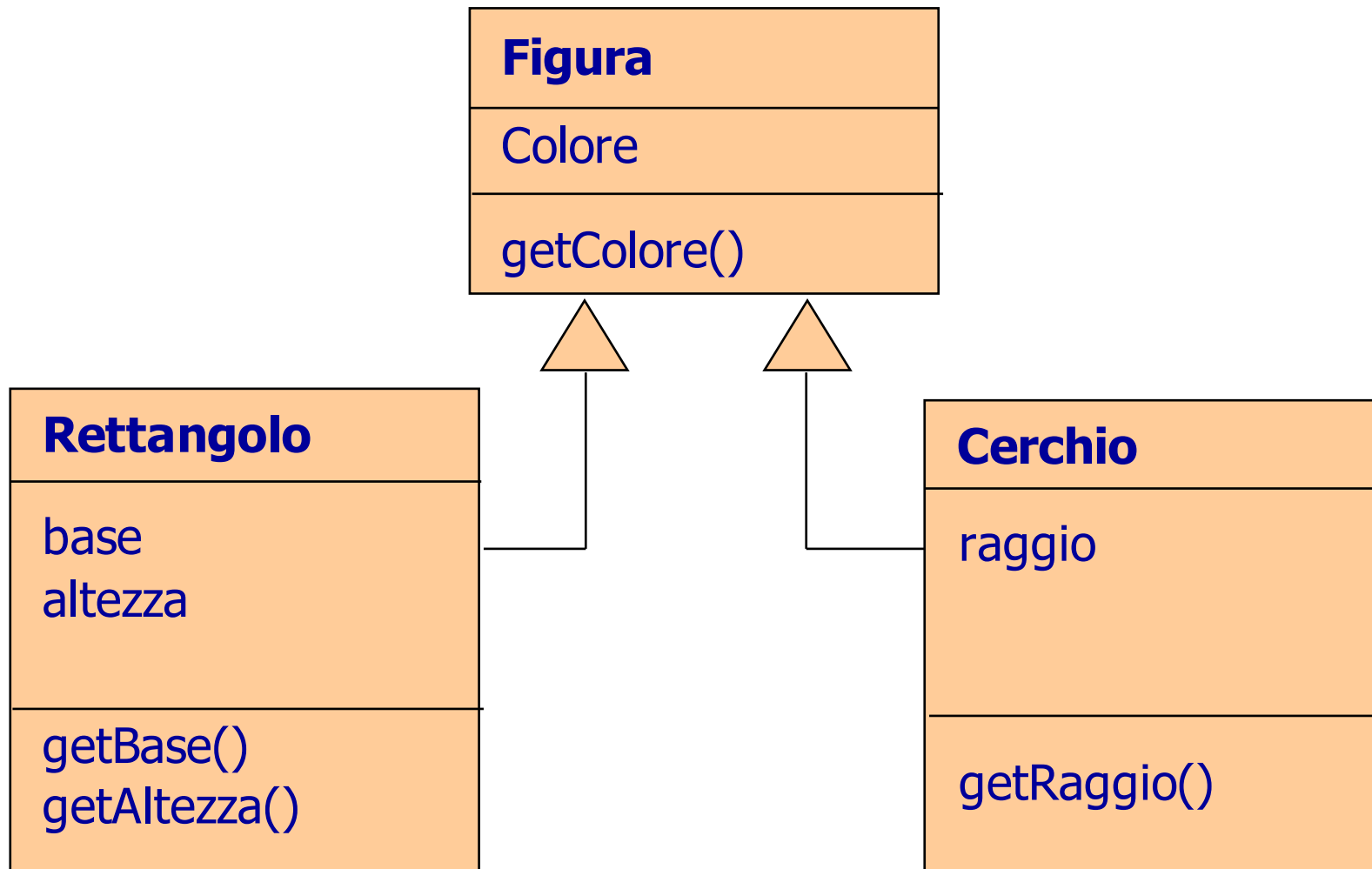
# Inizializzazione degli oggetti

---

- 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

---

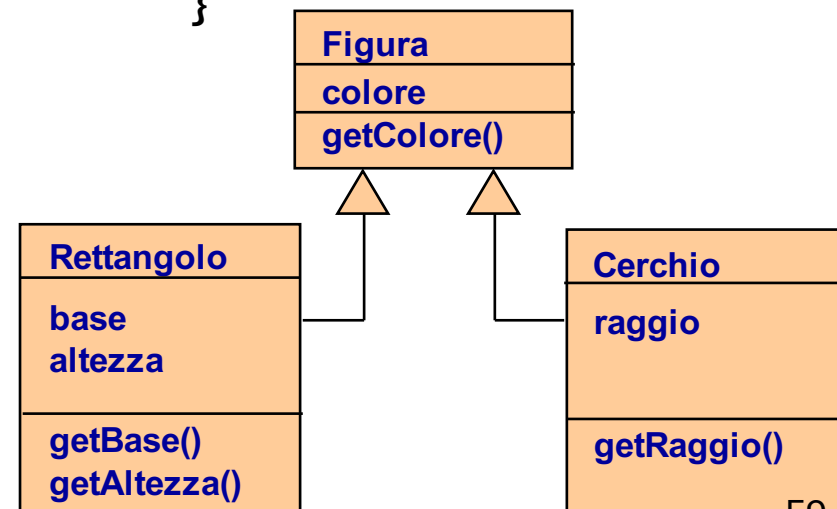


## Esempio (2)

```
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;  
    }  
}
```



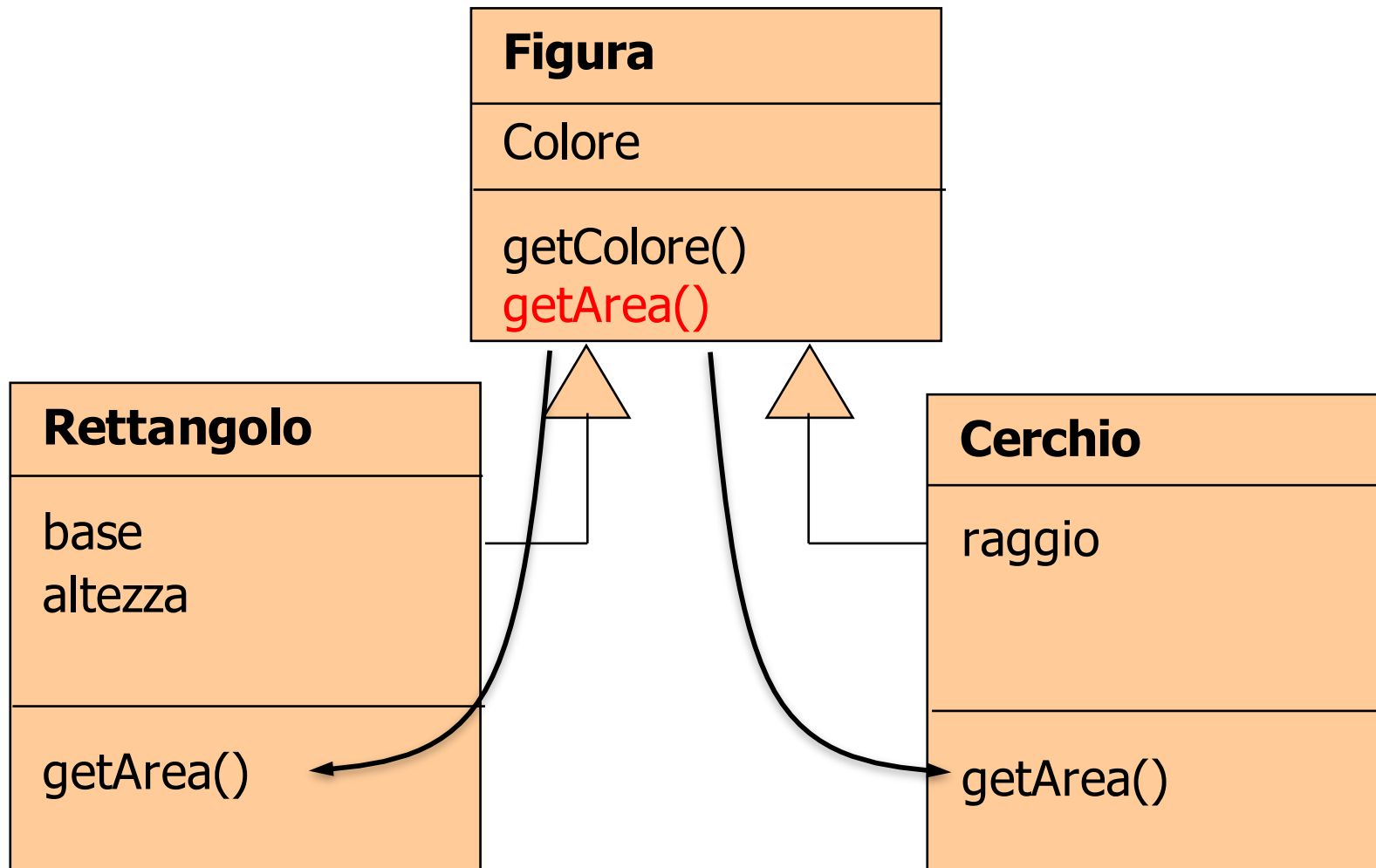


## Il problema dell'area

---

- 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



# 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 (stesso package)</b>	Si	Si	No	Si
<b>Altra Classe non sottoclasse (altro package)</b>	Si	No	No	No
<b>Sottoclasse (altro package)</b>	Si	No	No	Si



# 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`.





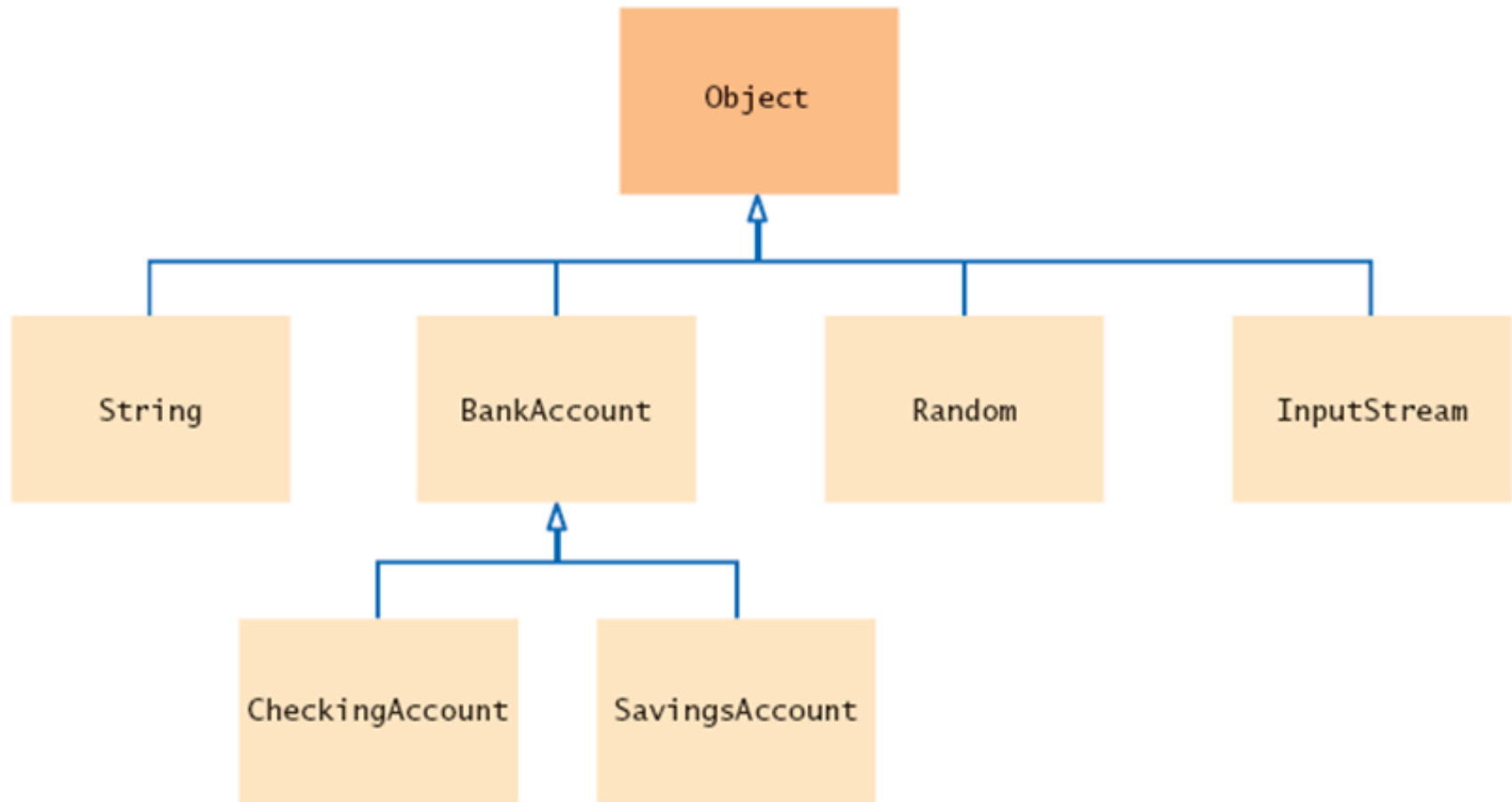
# Object: La classe universale

---

- 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

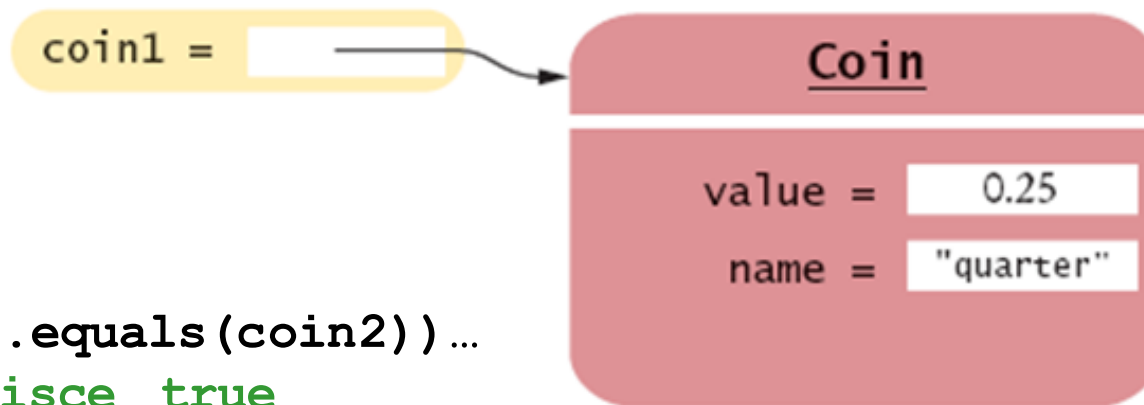
---

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

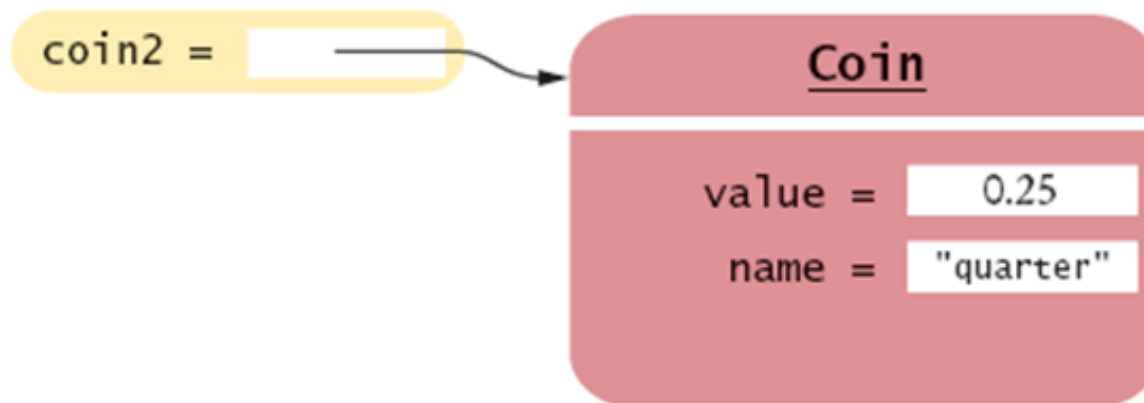
```
SavingsAccount sa = new SavingsAccount(10) ;  
System.out.println(sa) ;  
//stampa "SavingsAccount[balance=1000]  
           [interestRate=10]" ;
```

# Sovrascrivere `equals`

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

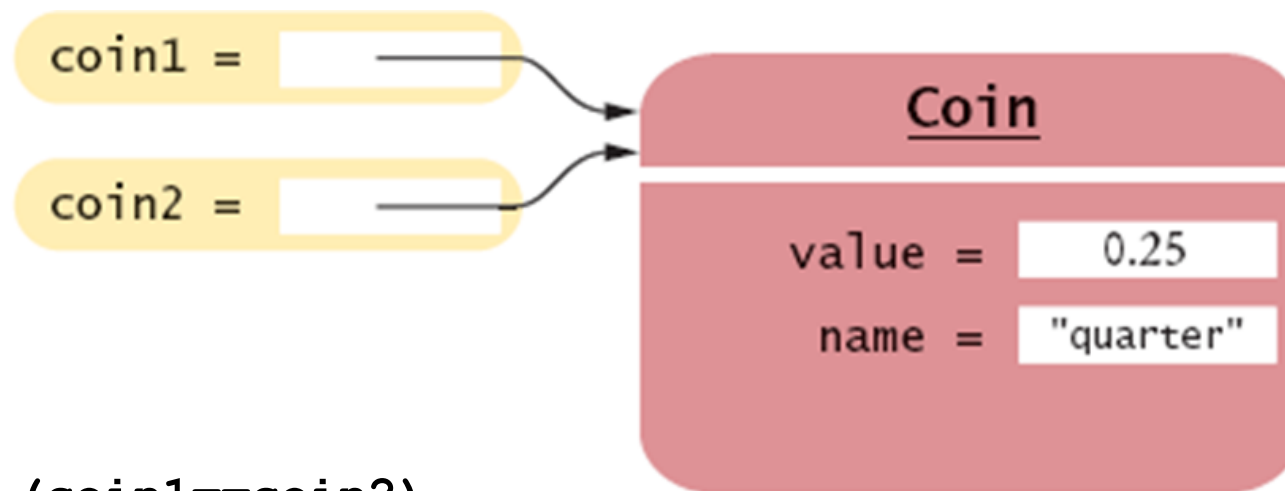


```
if (coin1.equals(coin2)) ...  
//restituisce true
```



# 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;  
}
```