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

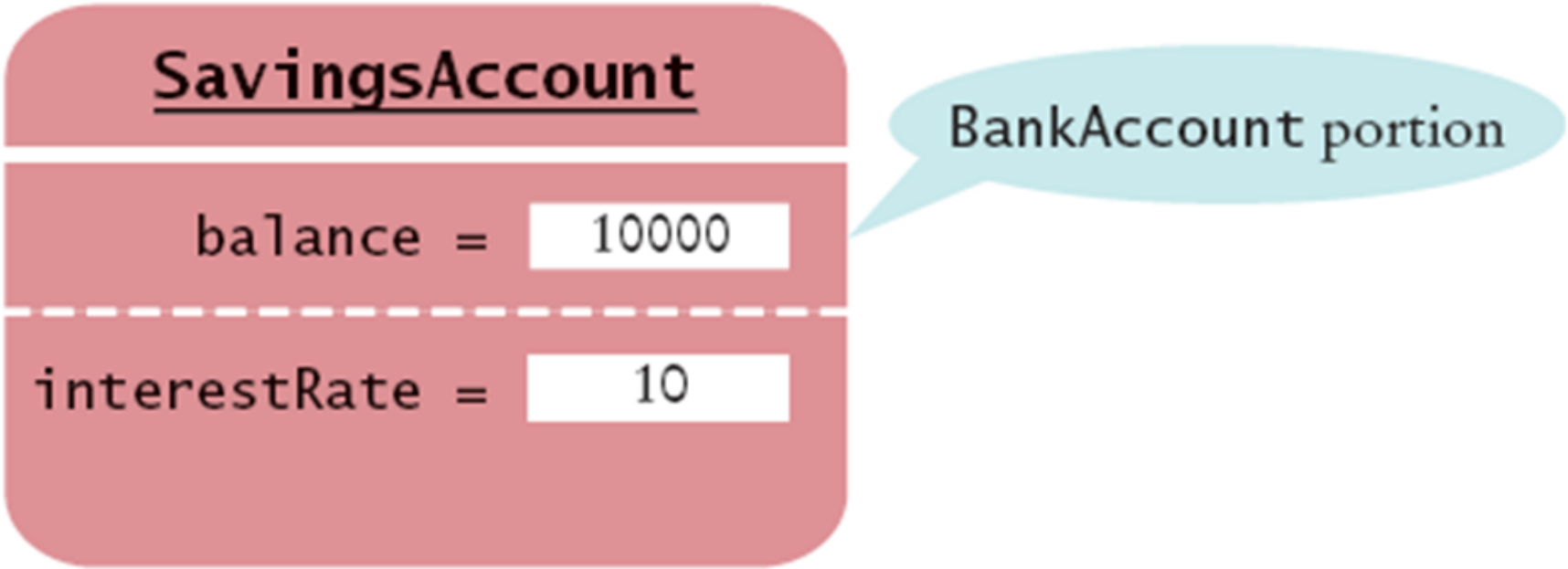
- La classe **`SavingsAccount`** eredita i metodi della classe **`BankAccount`**:
  - **`withdraw`**
  - **`deposit`**
  - **`getBalance`**

- 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

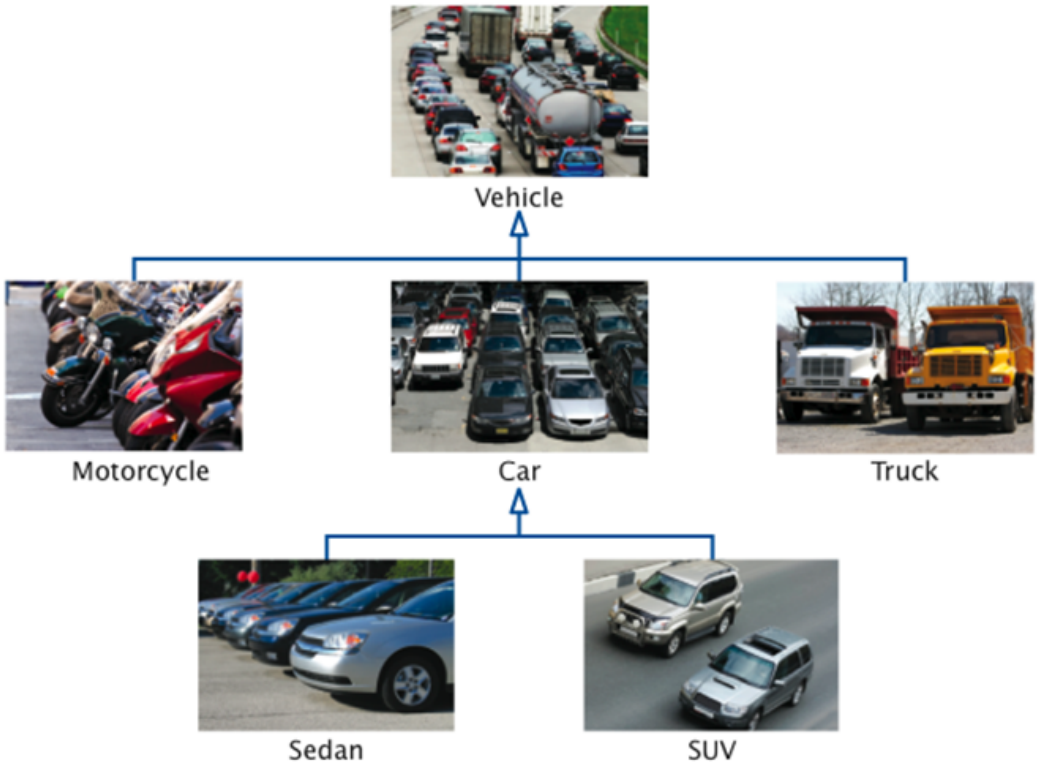
- **`SavingsAccount sa =`**
  `new SavingsAccount(10);`

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

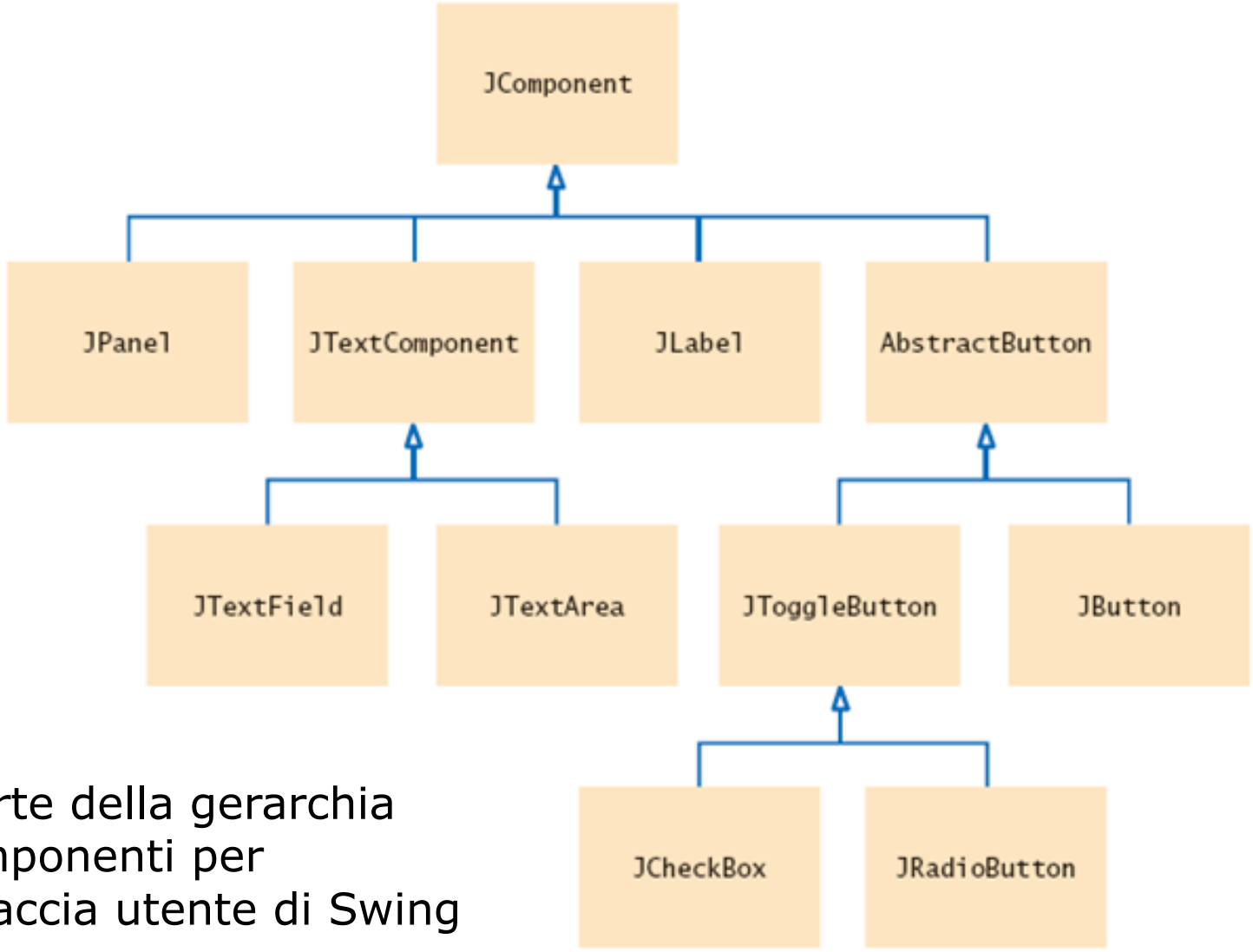
```
double interest = sa.getBalance()
        * sa.interestRate / 100;
sa.deposit(interest);
```

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



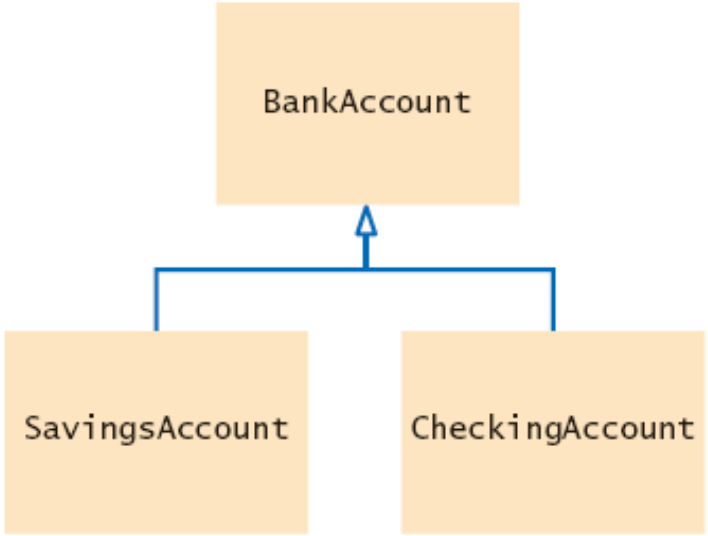
Gerarchie di ereditarietà



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

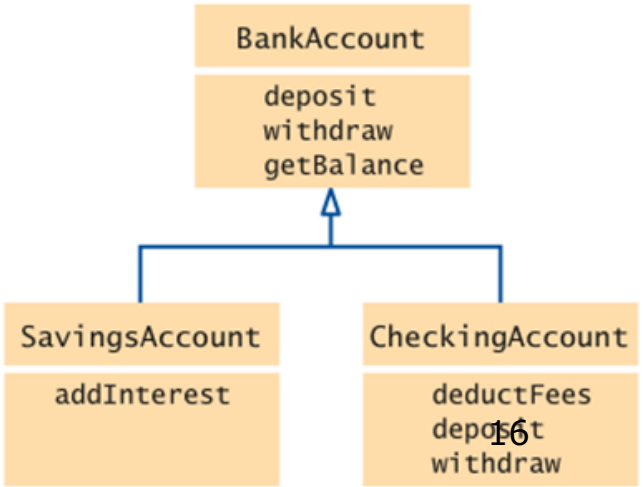
Gerarchie di ereditarietà

- 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**

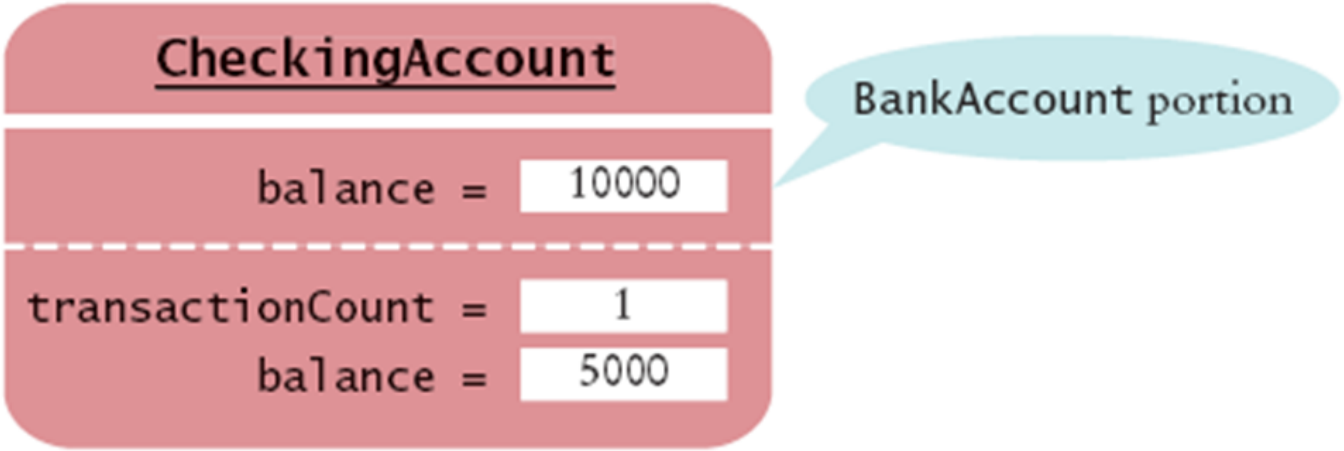
```
public void withdraw(double amount)
{

    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

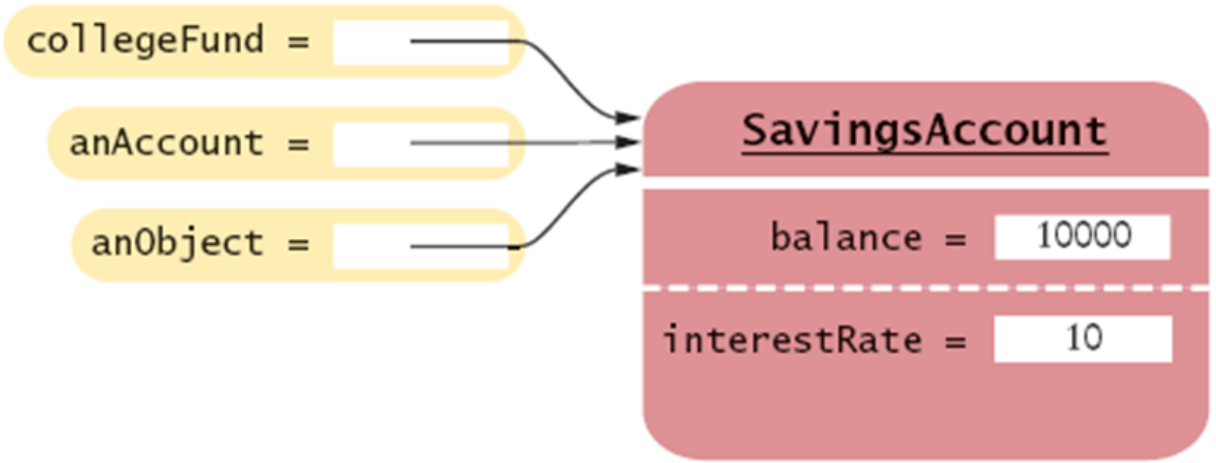
Conversione da Sottoclasse a Superclasse

- 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

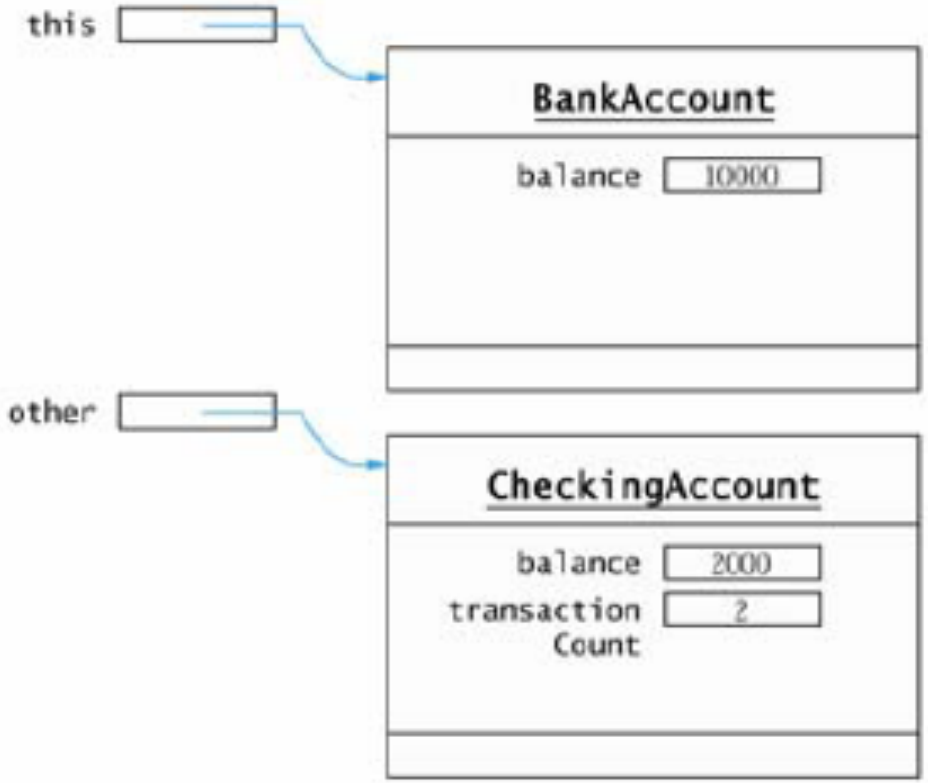
- E' lecito passare un riferimento di tipo **`CheckingAccount`** a un metodo che si aspetta un riferimento di tipo **`BankAccount`**

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

- l 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();
   ((CheckingAccoount) harrysChecking).deductFees();

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

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