Ereditarietà

Ereditarietà

- E' un meccanismo per estendere classi esistenti aggiungendo altri metodi e campi
  - raffinamento del concetto

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

- Si riutilizza codice:
  - tutti i metodi e le variabili d'istanza della classe BankAccount vengono ereditati automaticamente

Classe: SavingsAccount

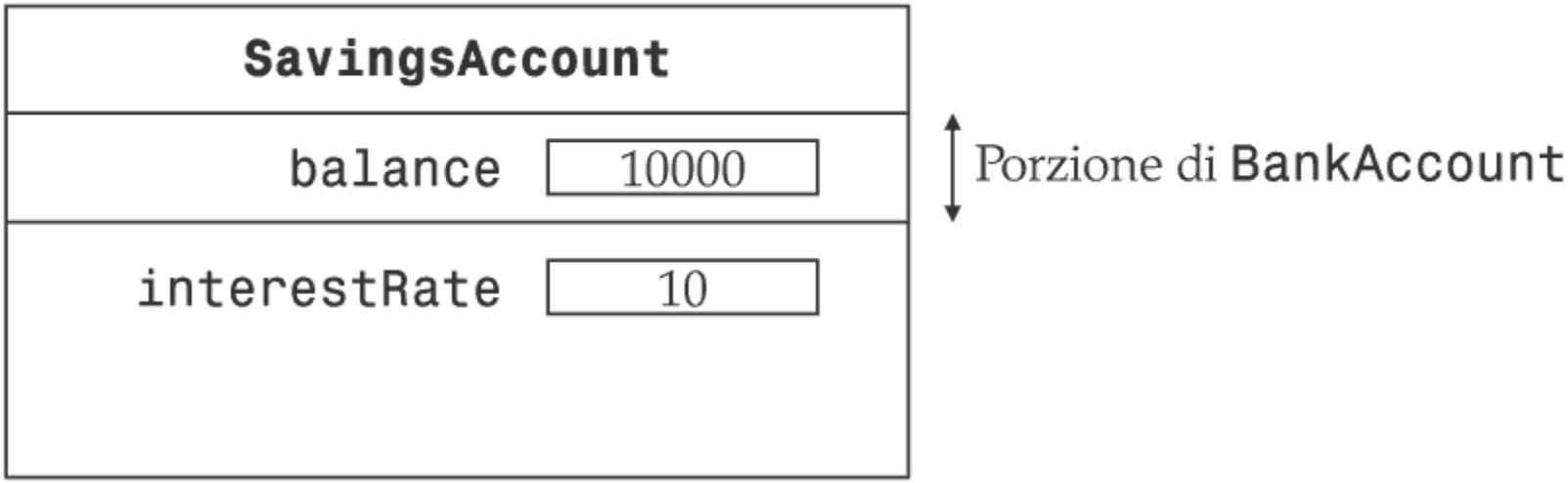
```
public class SavingsAccount extends BankAccount
   {
    public SavingsAccount(double rate){
       interestRate = rate;
    }
    public void addInterest(){
       double interest = getBalance()*interestRate/100;
       deposit(interest);
    }
    private double interestRate;
   }
```

- estende BankAccount aggiungendo variabile di istanza interestRate e metodo addInterest()

Osservazioni

- 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`**

Stati di oggetti SavingsAccount



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

SavingsAccount: metodo addInterest()

- 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);
```

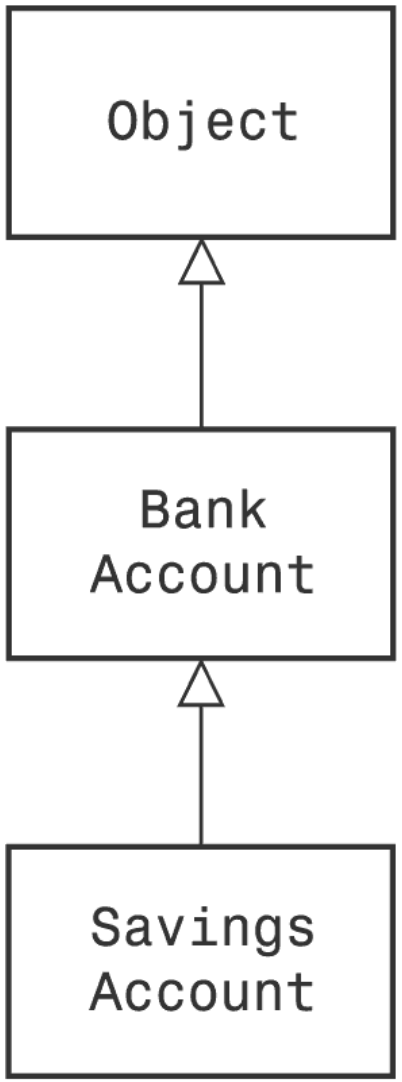
  - Non si può usare direttamente balance
    - è dichiarato private in BankAccount

Ereditarietà

- La classe pre-esistente (più generale) è detta SUPERCLASSE e la nuova classe (più specifica) è detta SOTTOCLASSE
  - **`BankAccount`**: superclasse
  - **`SavingsAccount`**: sottoclasse
- La superclasse definisce un supertipo del tipo della sottoclasse
- Gli oggetti della sottoclasse sono anche del tipo della superclasse

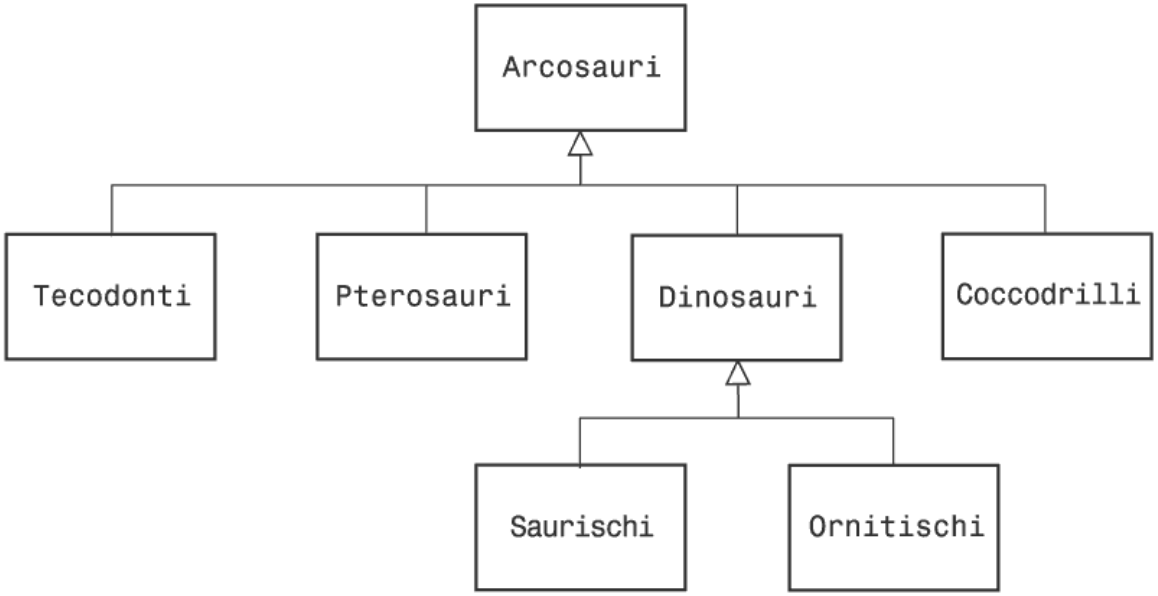
Classe Object

- 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 o)`
  - `Object clone()`

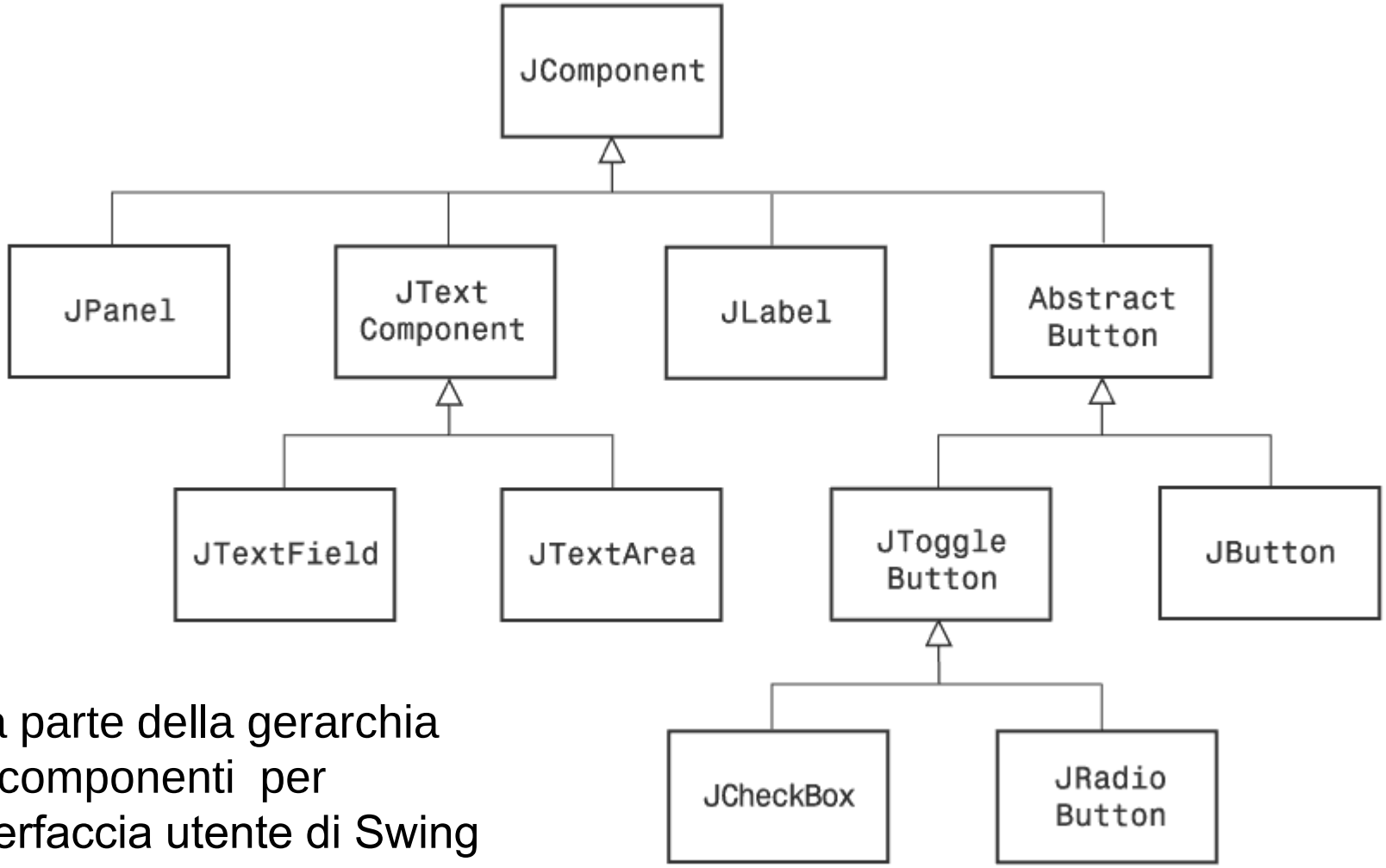


Gerarchia per ereditarietà

- In Java le classi sono organizzate in maniera gerarchica attraverso l'ereditarietà
  - Le classi che rappresentano concetti più generali sono più vicine alla radice
  - Le classi più specializzate sono alla fine delle diramazioni



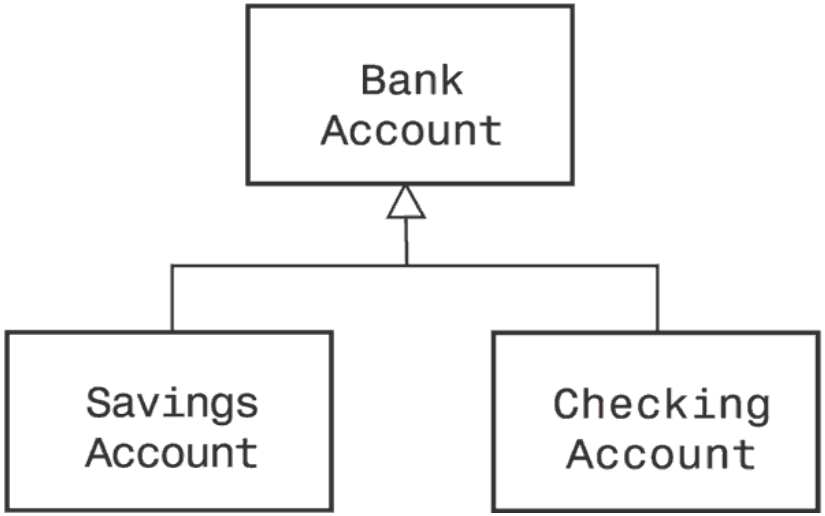
Gerarchia per ereditarietà



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

Gerarchia per 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



Gerarchia di BankAccount

- 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 i 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 un 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 di istanza
  - Le sottoclassi ereditano tutte le variabili di istanza della superclasse
- Definire nuove variabili di istanza
  - Esistono solo negli oggetti della sottoclasse
  - Possono avere lo stesso nome di quelle nella superclasse, ma non le sovrascrivono
    - 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

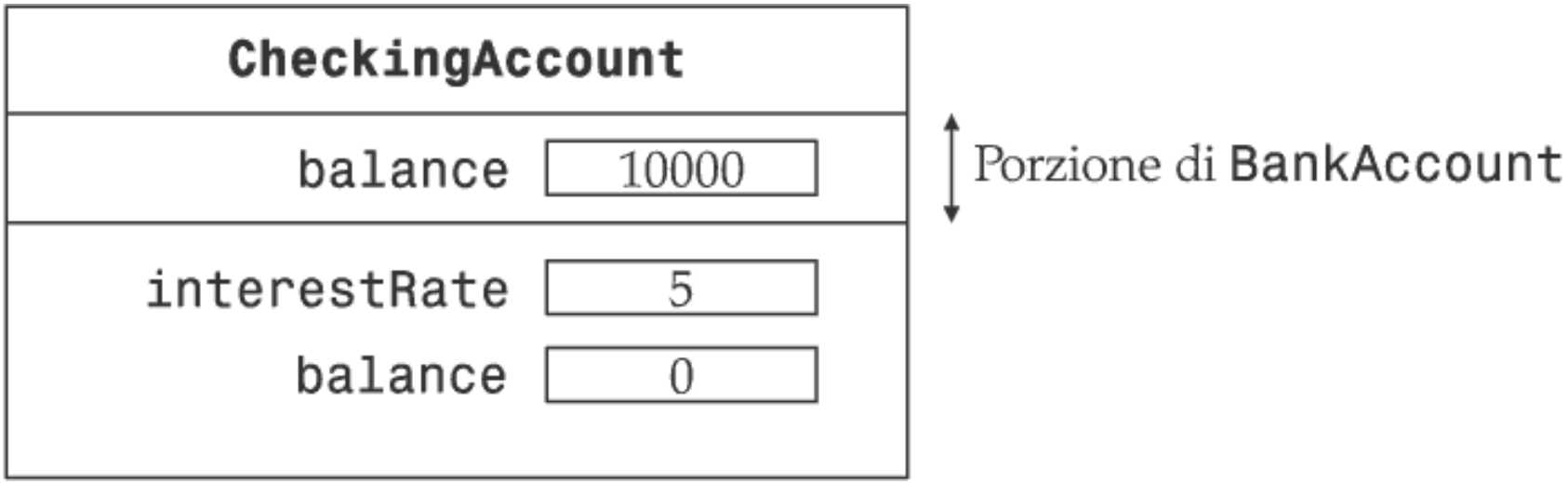
```
public void withdraw(double amount)
   {
      transactionCount++; // NUOVA VBL IST.

      //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);
     }
}
```

Mettere in ombra variabili di 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



Costruttore in 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;
   }
}
```

Costruttore in sottoclassi

- Se il costruttore della sottoclasse non chiama il costruttore della superclasse,

  viene invocato il costruttore predefinito

  della superclasse

- Ad es., se il costruttore di **`CheckingAccount`** non invoca il costruttore di **`BankAccount`**,

  viene impostato il saldo iniziale a zero

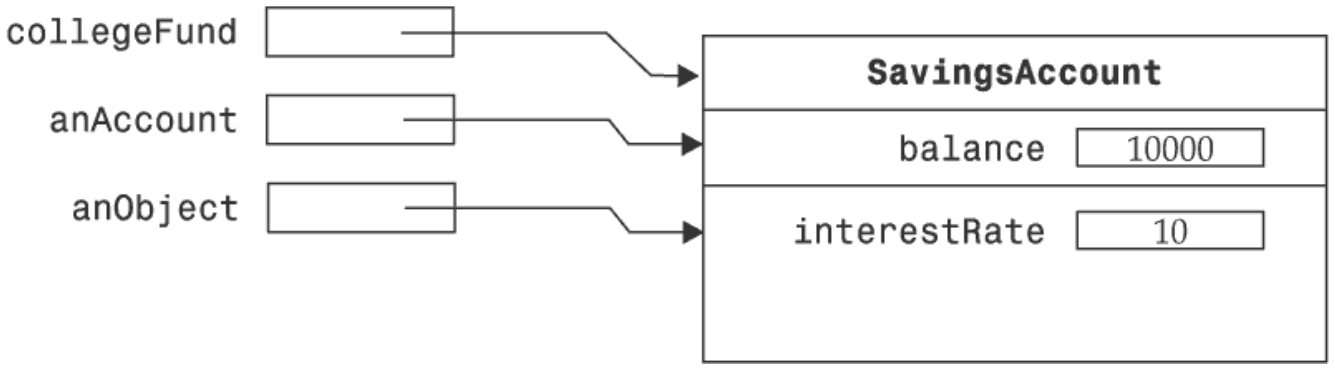
Conversione da Sottoclasse a Superclasse

- Si può assegnare un riferimento ad un oggetto di una sottoclasse in una variabile del tipo di 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

- Se si usa una variabile del tipo di una superclasse per un riferimento ad un oggetto della sottoclasse

  i metodi della sottoclasse non sono visibili:

```
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
```

Conversione da Superclasse a Sottoclasse

- Richiede casting
- Ha successo solo se oggetto è effettivamente un'istanza della classe

```
BankAccount collegeFund = ... ;
......
CheckingAccount x = (CheckingAccount)
collegeFund;
```

  - Ha successo solo se collegeFund contiene riferimento ad un oggetto di una classe di cui CheckingAccount è un supertipo

```
......
String x = (String) collegeFund;
```

  - Errore in fase di compilazione (non c'è relazione tra String e BankAccount)

Estensione e composizione

- Due modi fondamentali per costruire classi e riutilizzare codice

- Estensione:
  - Usa ereditarietà
  - Esprime una relazione di appartenenza della sottoclasse alla superclasse

- Composizione:
  - Aggrega oggetti di altre classi (definiti come variabili di istanza nella nuova classe)
  - Esprime una relazione di possesso (la classe composta ha istanze delle classi componenti)

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 void mioMetodo(...)`**
    - questo metodo non si può sovrascrivere

Come definire tipi di oggetti immutabili

- non fornire metodi modificatori
- rendere le variabili di istanza private
- impedire la definizione di sottoclassi
  - classe dichiarata final (semplice) oppure
  - costruttore private e istanziare oggetti attraverso metodi static (sofisticato)
- se le variabili di istanza contengono riferimenti a oggetti mutabili non consentire modifiche su questi oggetti:
  - non condividere i riferimenti fuori della classe (usare clonazione)

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

Accessibile da	`public`	`protected`	`package`	`private`
Stessa Classe	Si	Si	Si	Si
Altra Classe (stesso package)	Si	Si	Si	No
Sottoclasse (altro package)	Si	Si	No	No
Altra Classe non sottoclasse (altro package)	Si	No	No	No

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`**.

Classe astratta

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

- non può essere istanziata
- può avere metodi normalmente implementati ed altri *astratti*
  - Un metodo astratto non ha implementazione:

    ```
    public abstract void deductFees();
    ```
- non è obbligatorio avere metodi astratti

Classe astratta

- Le classi non astratte sono dette concrete
  - non possono contenere metodi astratti
- Le classi concrete che estendono una classe astratta sono OBBLIGATE ad implementarne i suoi metodi astratti
  - le classi astratte forzano la realizzazione di sottoclassi
- Un metodo astratto ha il vantaggio di
  - non dover scrivere un metodo fittizio che può essere ereditato dalle sottoclassi (senza obbligo di sovrascriverlo)
  - rendere il metodo parte dell'interfaccia pubblica della superclasse (facilita riutilizzo del codice)

Osservazioni su uso ereditarietà

- Raggruppare comportamenti comuni tra classi diverse in una classe astratta
  - Es. Personale con sottoclassi Dipendente, Quadro, Manager
- Fornire implementazioni alternative per uno stesso metodo (polimorfismo)
  - Es. calcolo paga per lavoratore a ore e lavoratore a stipendio fisso
- Raffinare l'implementazione di un metodo
  - Es. deposit di BankAccount e CheckingAccount
- Estendere i comportamenti di una classe esistente
  - Es. CollectibleCoin, MeasurableBankAccount.