Ereditarietà

Ereditarietà

- È un meccanismo per potenziare classi esistenti e funzionanti
- Quando vogliamo implementare una nuova classe ed è già disponibile una classe che rappresenta **un concetto più generale**, allora la nuova classe può ereditare da quella esistente

Ereditarietà

- Supponiamo, ad esempio, di voler definire una nuova classe **`SavingsAccount`** che rappresenta un conto bancario che garantisce un tasso di interesse fisso sui depositi
- Abbiamo già la classe **`BankAccount`** e un conto di risparmio è un caso speciale di conto bancario
- Utilizziamo l'ereditarietà per definire la nuova classe **`SavingsAccount`**

Ereditarietà: sintassi

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

- Tutti i metodi e le variabili istanza di **`BankAccount`** sono ereditati automaticamente dalla classe **`SavingsAccount`**

Ereditarietà: esempi

- Ad esempio possiamo usare il metodo **`deposit`** della classe **`BankAccount`** su un oggetto della classe **`SavingsAccount`**:

```
// Crea un conto di risparmio con
// tasso di interesse fisso del 10%:
SavingsAccount collegeFund = new
                        SavingsAccount(10);
// utilizzo il metodo deposit ereditato
// da BankAccount:
collegeFund.deposit(100);
```

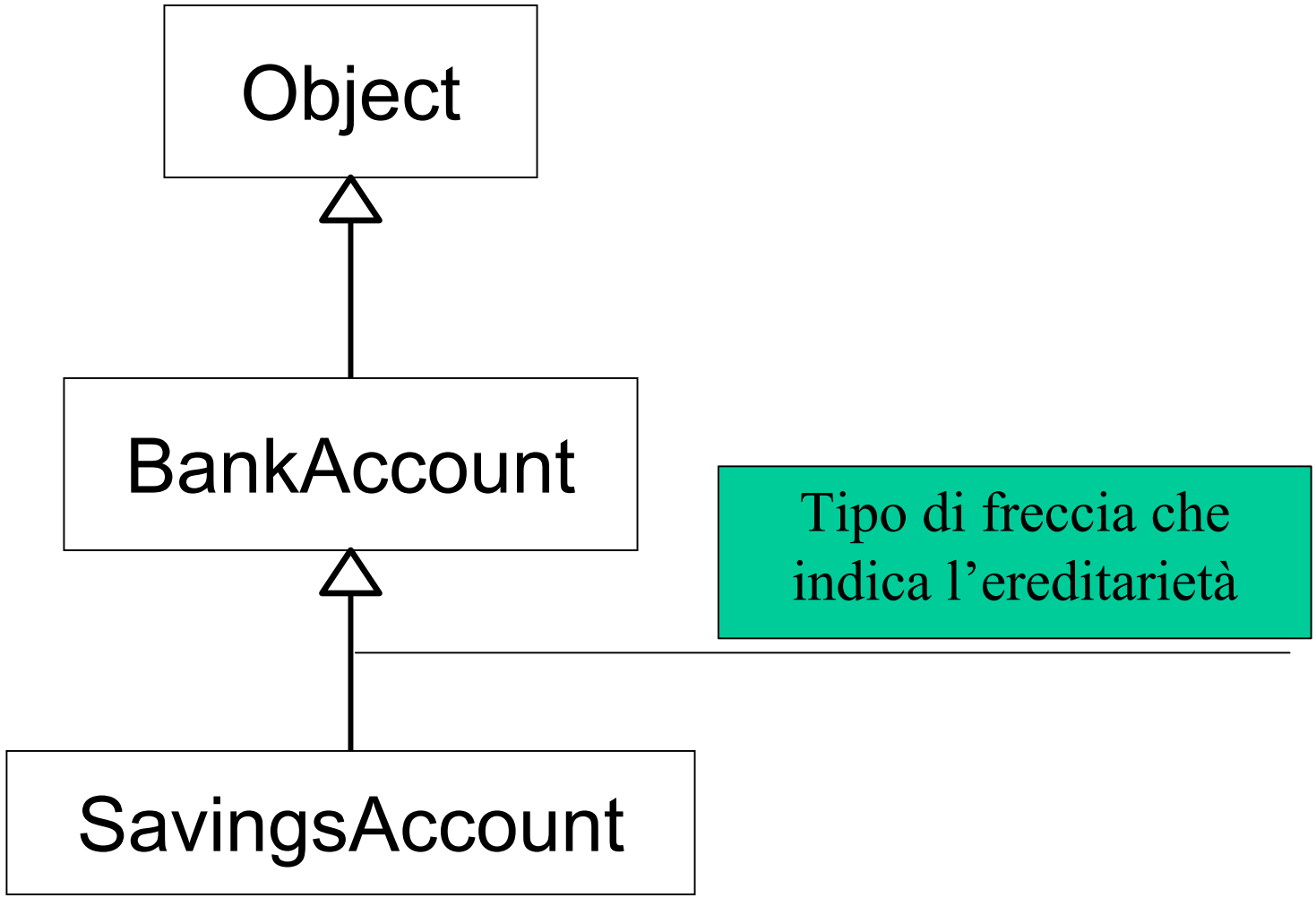
Ereditarietà: terminologia

- La classe più generica da cui si eredita viene detta **superclasse**
- La classe che eredita viene detta **sottoclasse**
- Nel nostro esempio quindi **`BankAccount`** è la superclasse e **`SavingsAccount`** è la sottoclasse

Ereditarietà: la classe Object

- In Java una classe che non estende esplicitamente un'altra classe è una sottoclasse della classe **`Object`**
- Ad esempio **`BankAccount`** è automaticamente una sottoclasse della classe **`Object`**
- La classe **`Object`** ha un numero limitato di metodi (vedere API) che vanno bene per tutti gli oggetti: **`toString`**, **`equals`**, **`clone`**, ...

Diagramma di classi UML



Ereditarietà vs implementazione di interfaccia

- Sono due concetti differenti
- Un'interfaccia non è una classe:
  - Non ha uno stato
  - Non ha un comportamento
  - Indica solamente un certo numero di metodi che bisogna implementare
- Una superclasse ha uno stato e un comportamento che vengono ereditati dalla sottoclasse

Vantaggi dell'ereditarietà

- Permette di modellare alcune situazioni in maniera più precisa
- Permette di **riutilizzare il codice**
- Non siamo costretti a rifare il lavoro di progettazione e messa a punto della nuova classe che già è stato fatto nella superclasse:
  - In **`SavingsAccount`** possiamo riutilizzare i metodi **`deposit`** **`withdraw`** e **`getBalance`** di **`BankAccount`** senza nemmeno toccarli

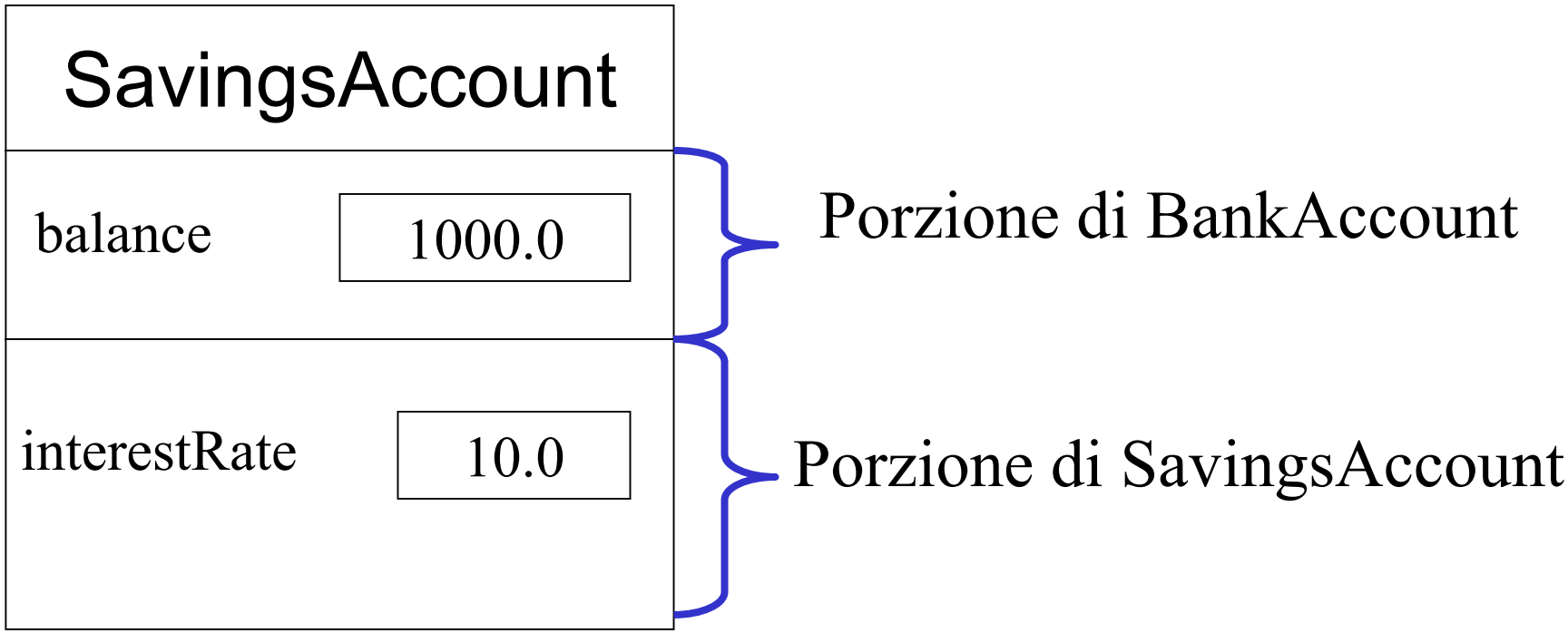
Contenuto della sottoclasse

- La sottoclasse può contenere:
  - Nuovi metodi
  - Nuove variabili istanza
  - Metodi della superclasse sovrascritti
- Nell'esempio abbiamo che in **`SavingsAccount`** dobbiamo aggiungere una variabile istanza per memorizzare il tasso di interesse e un nuovo metodo che aggiunge gli interessi maturati

SavingsAccount

```
public class SavingsAccount extends
                                    BankAccount
{
   public SavingsAccount(double rate)
   {  implementazione del costruttore
   }
   public void addInterest()
   {  implementazione del metodo
   }
   private double interestRate;
}
```

Rappresentaz. di un oggetto SavingsAccount



SavingsAccount: implementazione

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

Implicito
this.getBalance()

Implicito
this.deposit(interest)

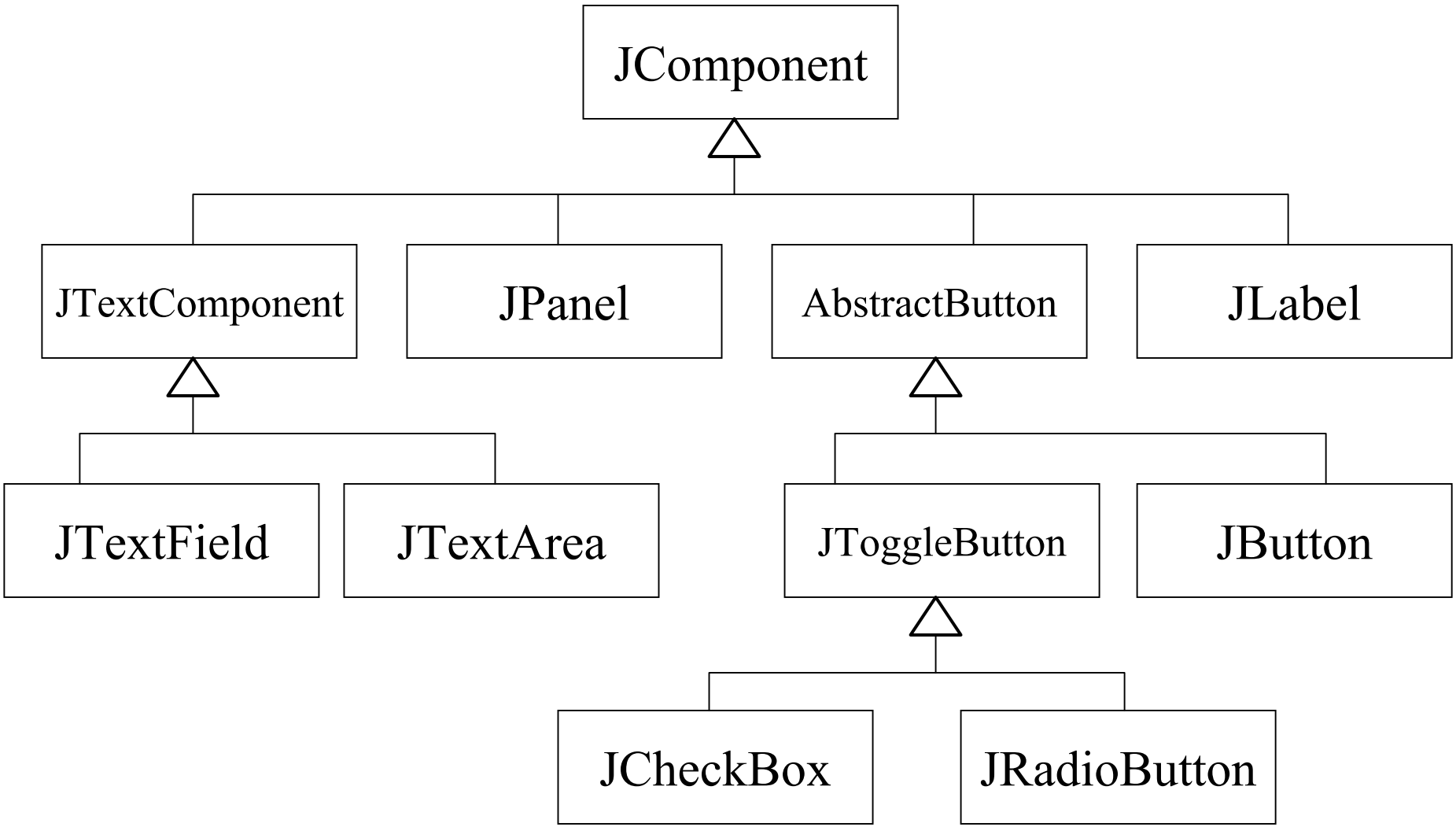
Giustificazione del nome

- Perché viene detta sottoclasse quella che eredita?
- La terminologia si riferisce a una interpretazione insiemistica:
  - I conti di risparmio sono particolari conti bancari, quindi l'insieme di tutti i possibili conti di risparmio è un **sottoinsieme** di tutti i possibili conti bancari
  - Il nome viene quindi dal fatto che la **sottoclasse** rappresenta un **sottoinsieme** degli oggetti possibili della superclasse

Gerarchie di ereditarietà

- Nel mondo reale spesso i concetti si classificano in gerarchie
- La stessa cosa si può fare in Java creando una gerarchia di classi (una classe = un concetto)
- I concetti più generali si trovano in alto nella gerarchia
- I concetti più specifici sono sottoclassi di classi più in alto nella gerarchia

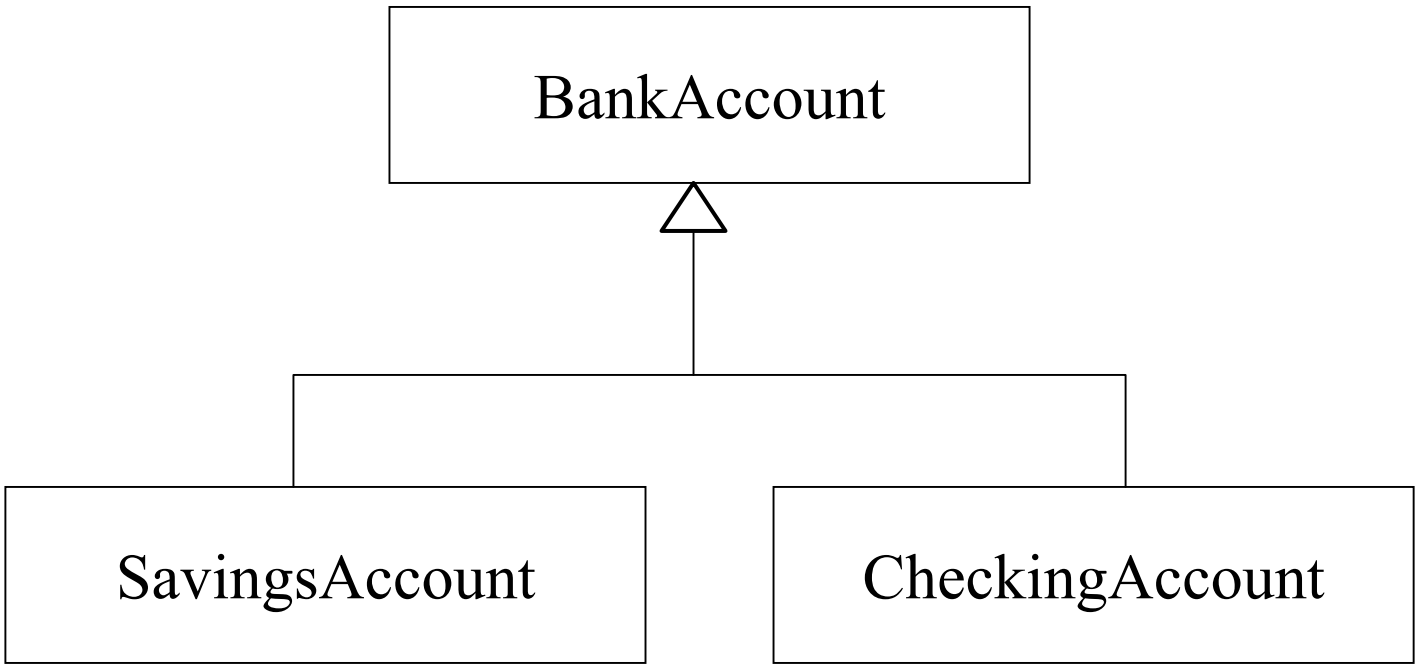
Esempio: gerarchia di oggetti grafici Swing



Un esempio semplice di gerarchia

- Un conto corrente è un conto bancario che non offre interessi, supporta un numero limitato di operazioni mensili gratuite ed addebita una commissione per tutte le altre
- Anche un conto corrente (**`CheckingAccount`**) può essere visto come un'estensione di **`BankAccount`**
- Otteniamo quindi la seguente gerarchia:

Un esempio semplice di gerarchia



Un esempio semplice di gerarchia

- Un **`CheckingAccount`** ha bisogno di un metodo **`deductFees`** che ogni mese addebita le commissioni e azzera un contatore delle operazioni
- I metodi **`deposit`** e **`withdraw`** vanno modificati per incrementare il contatore di operazioni
- Come facciamo?

Metodi delle sottoclassi

1. Possiamo **riscrivere** (**ridefinire**) un metodo della superclasse: il metodo deve avere lo stesso tipo di ritorno, lo stesso nome e la stessa sequenza di tipi di parametri (la cosiddetta *segnatura* o firma)
   - Quando il metodo ridefinito viene chiamato su un oggetto della sottoclasse il codice effettivamente eseguito è quello ridefinito
   - Questo accade anche se ci si riferisce all'oggetto della sottoclasse attraverso una variabile del tipo della superclasse (selezione posticipata)

Metodi delle sottoclassi

2. Possiamo ereditare metodi della superclasse: basta non riscriverli e sono ereditati automaticamente
   - Possono essere chiamati su oggetti della sottoclasse
   - Il codice eseguito è quello specificato nella superclasse

Metodi delle sottoclassi

3. Possiamo scrivere nuovi metodi: questi possono essere chiamati solo su oggetti della sottoclasse

Variabili istanza delle sottoclassi

- Le variabili istanza, a differenza dei metodi, non si possono sovrascrivere
- Possiamo:
  - Ereditare tutte le variabili istanza della superclasse (automatico)
  - Definire nuove variabili istanza che compariranno solo nello stato degli oggetti della sottoclasse

Variabili istanza delle sottoclassi

- Se ridefiniamo una variabile istanza (usando lo stesso nome e lo stesso tipo) allora
  - Gli oggetti avranno due variabili istanza con lo stesso nome e lo stesso tipo
  - La variabile della superclasse viene messa in ombra da quella della sottoclasse
  - I metodi della sottoclasse possono accedere solo alla variabile istanza della sottoclasse
- Pratica sconsigliata perché fonte di errori

Implementiamo CheckingAccount

- Dobbiamo aggiungere una variabile istanza intera **`transactionCount`** per contare il numero di operazioni e calcolare così il totale delle commissioni mensili
- Dobbiamo aggiungere un metodo **`deductFees`** che azzera il contatore, calcola le commissioni e le deduce dal saldo
- Dobbiamo sovrascrivere i metodi **`deposit`** e **`withdraw`** per far loro incrementare il contatore

Implementazione di CheckingAccount

- Problema:
- **Le variabili istanza private della superclasse, essendo private, non possono essere accedute dai metodi della sottoclasse!**
- Come possiamo fare?
- Nei metodi della sottoclasse è possibile usare la parola riservata **`super`** per chiamare metodi della superclasse sull'oggetto su cui si sta eseguendo il metodo

Riscrittura di deposit

```
public class CheckingAccount extends
                              BankAccount {
...
public void deposit(double amount) {
   transactionCount++;
   // chiamo il metodo della superclasse
   super.deposit(amount);
}
...
```

Errore comune

- Cosa sarebbe successo se non avvessimo specificato **`super.deposit(amount)`**, ma solo **`deposit(amount)`**?
- Il significato sarebbe stato, come da regola, quello di chiamare il metodo **`this.deposit(amount)`**
- Il metodo **`deposit`** da eseguire, poiché l'oggetto chiamato è della classe **`CheckingAccount`**, sarebbe stato quindi lo stesso!
- Sarebbe iniziata un sequenza infinita di chiamate allo stesso metodo che sarebbe terminata solo con un errore **`OutOfMemory`** dovuto al superamento della disponibilità di memoria per appilare le attivazioni relative alle chiamate!!!

Errore comune

- A volte si può essere tentati di ridefinire una variabile istanza perché vorremmo accedervi da un metodo ridefinito nella sottoclasse e, in questi casi, il compilatore non lo permette perché è una variabile privata della superclasse:

In **`SavingsAccount`**:

```
public void deposit(double amount) {
   transactionCount++;
   balance = balance + amount; // Errore
}
```

Errore comune

- Se si risolve l'errore ridichiarando **`balance`** si avrà che il metodo **`deposit`** farà riferimento alla nuova **`balance`** ridefinita nella sottoclasse (la variabile della superclasse viene messa in ombra)
- Tuttavia il metodo **`getBalance`**, che non viene ridefinito, restituirà il valore della variabile istanza **`balance`** della superclasse!!!

Costruttori della sottoclasse

- Così come tutte le classi anche le sottoclassi possono avere metodi costruttori
- Si hanno gli stessi problemi in caso di variabili private della superclasse: come fare per inizializzarle?
- Si ricorre allo stesso meccanismo: il costruttore della sottoclasse può chiamare un costruttore della superclasse tramite la chiamata **`super`**(*parametri*)
- Questa chiamata deve **obbligatoriamente** essere la prima riga di codice del costruttore

Costruttori della sottoclasse

```
...
public SavingsAccount(double amount,
                      double rate) {
   super(amount); // Inizializza il saldo
   interestRate = rate;
}
...
```

Costruttori ridefiniti

- Un costruttore della sottoclasse può anche ridefinirne uno della superclasse
- Il costruttore ridefinito avrà nome diverso (quello della sottoclasse), ma lo stesso tipo di parametri
- Anche in questo caso il corrispondente costruttore della superclasse può essere invocato nella prima riga di codice del nuovo costruttore

Costruttori ridefiniti

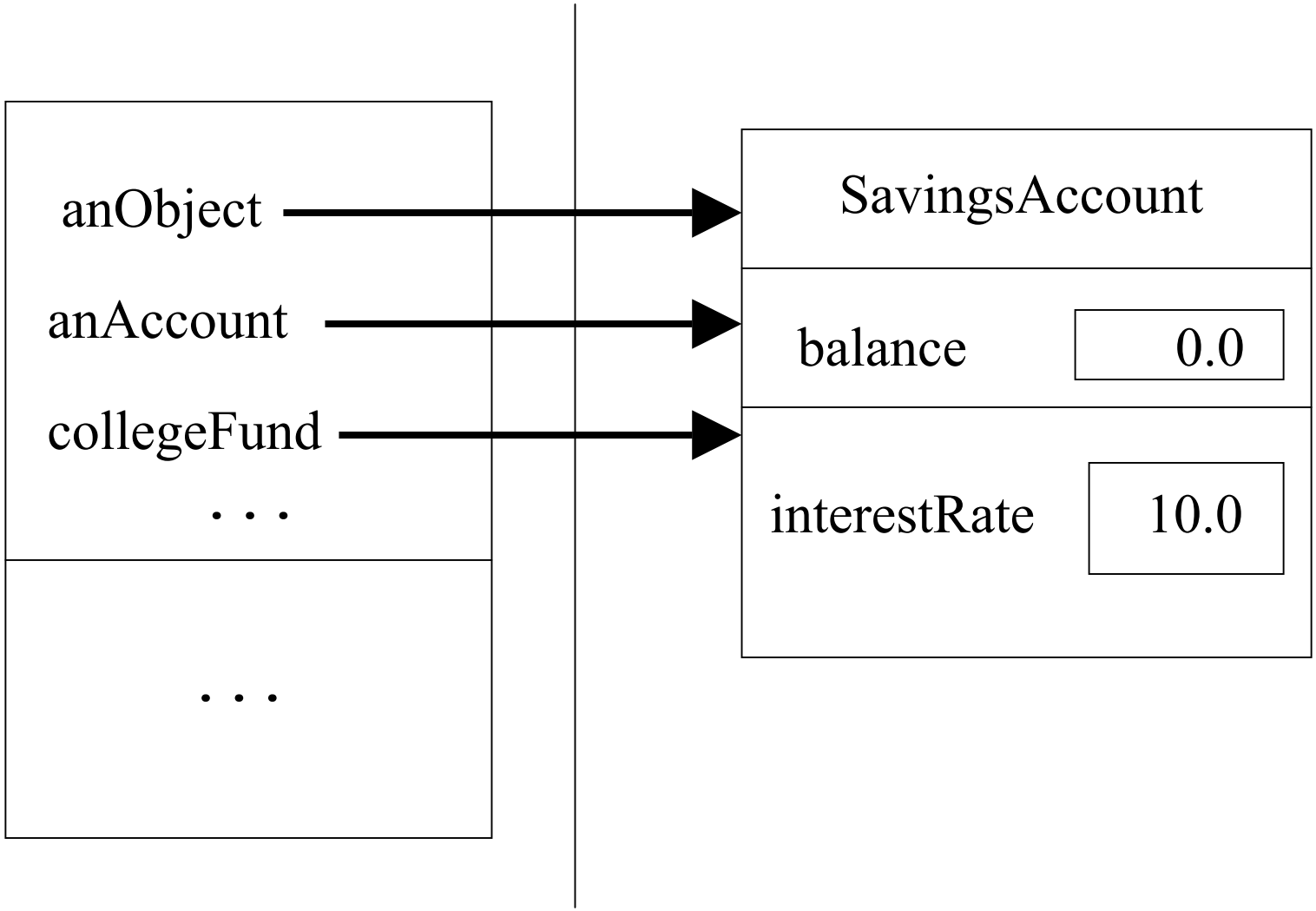
```
public class CheckingAccount extends
                                    BankAccount {
  public CheckingAccount(double
                          initialBalance) {
    // Costruisce la superclasse
    super(initialBalance);
    // Inizializza il contatore di
    // operazioni
    transactionCount = 0;
  }
...
```

Conversione da sottoclasse a superclasse

- Abbiamo già visto questo meccanismo con le interfacce
- Un oggetto di una sottoclasse può essere sempre visto come un oggetto della superclasse

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

Conversione da sottoclasse a superclasse



Chiamate di metodi

- Attraverso il riferimento **`collegeFund`** si possono chiamare tutti i metodi della classe **`SavingsAccount`** (inclusi ovviamente quelli ereditati)
- Attraverso il riferimento **`anAccount`**, invece, si possono chiamare solo i metodi di **`BankAccount`**
- Infine, attraverso **`anObject`**, possono essere chiamati solo i metodi della classe **`Object`**

Quando usare la conversione

- È molto utile usare la conversione di tipo da sottoclasse a superclasse perché ciò permette di riutilizzare codice già scritto
- Ad esempio, supponiamo di aver scritto, per **`BankAccount`** un metodo transfer:

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

Quando usare la conversione

- Possiamo utilizzare lo stesso metodo **`transfer`** anche per fare trasferimenti tra conti di **`SavingsAccount`** e di **`CheckingAccount`**

```
SavingsAccount momsSavings =
          new SavingsAccount(10000);
CheckingAccount harrysAccount =
          new CheckingAccount(100);
momsSavings.transfer(1000,
                     harrysAccount);
```

Selezione posticipata e polimorfismo

- Abbiamo già visto che i metodi effettivamente eseguiti nelle chiamate sono quelli della classe reale a cui l'oggetto che viene chiamato appartiene
- Questo è il meccanismo della **selezione posticipata**
- Naturalmente se il metodo chiamato è ereditato dalla superclasse allora verrà eseguito comunque il metodo della superclasse (differenza con le interfacce, che non hanno implementazioni di metodi!)
- Il **polimorfismo** è invece la capacità di variabili o parametri, ad esempio **`other`**, di riferire oggetti di tipi diversi comportandosi in maniera diversa a seconda dei casi

Classi astratte

- Con il meccanismo dell'ereditarietà una sottoclasse può ereditare alcuni metodi della superclasse e può ridefinirne altri
- Esiste un meccanismo per obbligare le sottoclassi a ridefinire un metodo
- Può essere utile quando non esiste una buona impostazione predefinita per la superclasse
- Si possono dichiarare uno o più metodi della classe come **astratti**

Classi astratte

- Ad esempio supponiamo che la direzione di una banca con molte filiali decida che ogni conto debba avere delle deduzioni mensili per vari tipi di commissioni

```
public class BankAccount {
   public void deductFees() {
      // ??
   }
   ...
}
```

Classi astratte

- Naturalmente ogni filiale della banca potrà decidere la portata di queste deduzioni a seconda del proprio rapporto con i clienti
- Quindi cosa va scritto nell'implementazione del metodo?
- Sarebbe comodo poter delegare l'implementazione del proprio metodo ad ogni filiale
- Potrebbe essere un errore implementarlo con una deduzione pari a zero: se poi una filiale si dimentica di ridefinirlo?

Classi astratte

- Diciamo esplicitamente che il metodo nella classe non ha un'implementazione e che quindi, se si vuole creare un oggetto, si deve estendere la classe e fornire un implementazione per il metodo:

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

Classi astratte

- Una classe che ha almeno un metodo **`abstract`** deve essere dichiarata **`abstract`**
- Non si possono creare oggetti di una classe astratta!
- Per farlo bisogna estendere la classe e implementare almeno tutti i metodi astratti
- Le classi astratte differiscono dalle interfacce: possono avere sia variabili istanza sia alcuni metodi implementati (detti anche concreti)

Classi non estensibili

- In alcuni casi invece di obbligare i programmatori delle sottoclassi a implementare certi metodi si può volere proprio il contrario: cioè si vuole impedire che una certa classe possa essere estesa
- Per ottenere ciò basta aggiungere alla definizione della classe la parola riservata **`final`**:

```
public final class MyFinalClass {
  ... }
```

Classi non estensibili

- Ad esempio la classe **`String`** è dichiarata **`final`**
- La parola **`final`** può essere abbinata anche solo a certi metodi di una classe
- La classe potrà essere estesa, ma i metodi **`final`** non potranno essere ridefiniti

```
public class AccessoRemoto {
   ...
   public final boolean
          checkPassword(String password) {
...
} }
```

Acesso protected

- Oltre a **`public`** e **`private`** esiste in Java anche un altro specificatore di accesso: **`protected`**
- Una variabile istanza o un metodo **`protected`** ha possibilità di accesso che sono a metà strada tra **`public`** (accesso a tutti) e **`private`** (accesso solo ai metodi della classe: accesso negato perfino alle sottoclassi!)
- Una variabile istanza o un metodo **`protected`** possono essere acceduti **dalle sottoclassi e da tutte le classi nello stesso pacchetto**

Accesso protected

- L'uso dell'accesso **`protected`** risolverebbe il problema dell'accesso alle variabili private della superclasse da parte della sottoclasse
- **Ma** la possibilità di accesso a tutte le classi dello stesso pacchetto apre falle troppo grandi nella sicurezza perché **chiunque può aggiungere una sua classe ad un pacchetto** ed ottenere così l'accesso alle variabili istanza e ai metodi protected!

Accesso non specificato

- Se non si specifica nessuno specificatore di accesso per classi, variabili istanza e metodi il compilatore Java assegna di default l'accesso di pacchetto: tutte le classi dello stesso pacchetto possono accedere
- È una scelta giusta per le classi, ma per gli stessi motivi visti riguardo l'accesso **`protected`**, non è una buona idea per variabili istanza e metodi

In linea generale

- Quindi il tipo di accesso da preferire sempre per le variabili istanza è il privato
- Nel caso dei metodi invece si può considerare anche il caso di definirli **`protected`** per limitare il loro uso solo alle classi del pacchetto e alle sottoclassi
- Ogni volta che si esce da questo schema è opportuno riflettere adeguatamente sulla scelta fatta.