Interfacce, Polimorfismo, Ereditarietà

- Interfacce
- Polimorfismo
- Ereditarietà

DataSet

- Consideriamo la classe **`DataSet`**
- Semplicemente si può aggiungere ad un oggetto **`DataSet`** dei valori **`double`** con un metodo **`add`**
- Si può poi chiedere in qualunque momento la media dei valori o il valore massimo inserito fino a quel momento
- Questo tipo di oggetto potrebbe essere utile in generale, cioè potrebbe calcolare le stesse cose ma non solo per valori double, ma anche per oggetti di generiche classi

Interfaccia pubblica di DataSet

```
public class DataSet {
  public DataSet();
  public void add(double x);
  public double getMinimum();
  public double getMaximum();
  public double getAverage();
}
```

- Scrivere l'implementazione

DataSet per BankAccount

- Potremmo voler creare dei **`DataSet`** per oggetti della classe **`BankAccount`**

```
public class DataSetBankAccount {
...
  public void add(BankAccount x) {
    sum = sum + x.getBalance();
    if (count == 0 ||
        maximum.getBalance() <
        x.getBalance())
      maximum = x;
  count++;
} ...
```

DataSet per BankAccount

```
...
public BankAccount getMaximum() {
   return x;
}
...
private BankAccount maximum;
...
}
```

DataSet per Coin

- Potremmo anche voler implementare un **`DataSet`** che ci fornisca sempre lo stesso servizio, ma stavolta per la classe **`Coin`**:

```
public class DataSetCoin {
...
  public void add(Coin x) {
    sum = sum + x.getValue();
    if (count == 0 ||
        maximum.getValue() <
        x.getValue())
      maximum = x;
  count++;
} ...
```

DataSet per Coin

```
...
public Coin getMaximum() {
   return x;
}
...
private Coin maximum;
...
}
```

DataSet

- Il meccanismo usato dalle varie classi **`DataSet`** è lo stesso in tutti i casi: cambiano solo i dettagli
- In tutti i casi gli oggetti che vengono aggiunti al **`DataSet`** hanno un certo valore, nel loro stato, che viene usato per calcolare la media e il massimo
- Tutte le classi in questione potrebbero accordarsi su un unico metodo **`getMeasure`** che dia per ogni oggetto il valore da considerare per il **`DataSet`** (ogni classe decide cioè il valore da considerare "misura" dei suoi oggetti)

DataSet

- Lo schema del metodo **`add`** diventerebbe quindi:

```
sum = sum + x.getMeasure();
if (count == 0 ||
    maximum.getMeasure() <
    x.getMeasure())
  maximum = x;
  count++;
```

DataSet

- Ma quale dovrebbe essere il tipo della variabile **`x`**?
- In linea di principio **`x`** potrebbe essere un oggetto di una **qualunque** classe che renda disponibile un metodo **`getMeasure`**!
- Questo tipo di situazione si può modellare in Java tramite il meccanismo delle **`interface`**

Interfacce

- Non facciamo confusione tra l'interfaccia pubblica di una classe, che abbiamo visto essere l'insieme dei suoi metodi e variabili istanza pubbliche, e un' interfaccia dichiarata in questo modo:

```
public interface Measurable {
   double getMeasure();
}
```

Interfacce

- Questa dichiarazione dichiara semplicemente un "**contratto**"
- Il contratto **`Measurable`** dice che per essere rispettato da una certa classe c'è bisogno che essa fornisca un metodo di nome **`getMeasure`**, senza parametri, che restituisca un **`double`**
- Un'interfaccia **non è una classe**: non si possono creare oggetti di tipo **`Measurable`**!

Interfacce

- Tutti i metodi di un'interfaccia non hanno l'implementazione: sono metodi astratti
- Tutti i metodi di una interfaccia, inoltre, sono automaticamente pubblici
- Una interfaccia non può avere variabili istanza

Però:

- Possiamo usare un' interfaccia come un nome di classe qualsiasi nei nostri programmi

DataSet

```
public class DataSet {
...
  public void add(Measurable x) {
    sum = sum + x.getMeasure();
    if (count == 0 ||
        maximum.getMeasure() <
        x.getMeasure())
      maximum = x;
  count++;
} ...
```

DataSet

```
...
public Measurable getMaximum() {
   return x;
}
...
private Measurable maximum;
...
}
```

Uso di DataSet

- Una classe come **`DataSet`** che utilizza il nome di una interfaccia come può essere usata?
- Si possono passare, ai metodi che richiedono parametri di tipo **`Measurable`**, oggetti **qualsiasi** di una classe che **implementi** (realizzi) l'interfaccia **`Measurable`**

Implements

- La sintassi per indicare che una certa classe implementa (o realizza) una certa interfaccia è la seguente:

```
public class NomeClasse implements
  NomeInterfaccia {
/* Solita definizione di classe in
  cui però devono apparire, con
  l'implementazione, tutti i metodi
  dichiarati nell'interfaccia, con gli
  stessi parametri nello stesso
  ordine */
...
}
```

Implements

- Ad esempio possiamo far implementare l'interfaccia **`Measurable`** alla classe **`BankAccount`**

```
public class BankAccount implements
                          Measurable {
...
  public double getMeasure(){
     return balance;
  }
...
```

Implements

- Ma anche alla classe Coin!

```
public class Coin implements
                          Measurable {
...
  public double getMeasure(){
     return value;
  }
...
```

Implements

- Possiamo far implementare l'interfaccia **`Measurable`** a tutte le classi che vogliamo
- Un oggetto della classe **`DataSet`** potrà ricevere oggetti qualunque di queste classi (anche mischiati)

Interfacce

- Una classe può implementare anche più di una interfaccia
- In questo caso basta elencare, separate da virgola, tutte le interfacce che implementa dopo la parola riservata **`implements`**
- I metodi della classe che corrispondono a quelli dell'interfaccia implementata devono obbligatoriamente essere dichiarati **`public`**
- Un'interfaccia va definita in un file .java che si chiama con lo stesso nome dell'interfaccia (esattamente come per le classi pubbliche)

Test

- Consultare il Codice allegato DataSetPolimorfoTest.java
- Vediamo che variabili dichiarate di tipo **`Measurable`**, come ad esempio **`max`**, possono contenere puntatori a oggetti di classi diverse
- L'importante è che ognuna di queste classi implementi l'interfaccia **`Measurable`**
- I riferimenti in variabili dichiarate di tipo **`Measurable`** possono essere usati solo per chiamare metodi dell'interfaccia **`Measurable`**, anche se in realtà sull'oggetto puntato potrebbero essere chiamati anche altri metodi (ad esempio se è un oggetto di **`BankAccount`**)

Conversione dei tipi e cast

- Osserviamo la riga di codice:

```
bankData.add(new BankAccount(10000));
```

- Passiamo il riferimento ad un oggetto di tipo **`BankAccount`** a un metodo che ha un parametro di tipo **`Measurable`**
- Ciò è lecito perché abbiamo fatto in modo che la classe **`BankAccount`** implementasse l'interfaccia **`Measurable`**
- Conversioni di questo tipo possono essere effettuate automaticamente

Conversione dei tipi e cast

- È lo stesso principio di quando assegnamo un **`int`** a un **`double`**: non c'è perdita di informazione e quindi l'assegnamento può essere fatto senza problemi
- Qui l'effetto della conversione consiste nel mascherare alcune potenzialità dell'oggetto facendo in modo che possano essere utilizzate solo le funzionalità che sono specificate nell'interfaccia

Conversione dei tipi e cast

- Crea un oggetto della classe **`BankAccount`**:

```
BankAccount account = new
  BankAccount(10000);
```

- Conversione legittima perché **`BankAccount`** implementa **`Measurable`**:

```
Measurable x = account;
```

- Su **`x`** però posso chiamare solo i metodi dichiarati in **`Measurable`**

```
x.deposit(100); // Errore!!
```

Conversione dei tipi e cast

```
System.out.println(x.getMeasure());
```

- Legittimo perché **`x`** è di tipo **`Measurable`**
- Comunque l'oggetto puntato da **`x`** è ancora un oggetto **`BankAccount (x == account)`**:

```
account.deposit(100); // Ok
```

- Posso creare anche un oggetto della classe **`Coin`** e assegnarlo a **`x`**:

```
Coin dime = new Coin(0.1, "Dime);
x = dime; // legittimo
```

Conversioni di tipo e cast

- In generale quando si ha un riferimento di tipo **`Measurable`** non si conosce il tipo vero dell'oggetto a cui punta il riferimento (nessun oggetto può essere creato dall'interfaccia **`Measurable`**! Solo da classi che la implementano)
- Se però si conosce, ad esempio dal contesto del programma, il vero tipo dell'oggetto riferito da una variabile di tipo **`Measurable`** si può fare un **cast esplicito**

Conversioni di tipo e cast

```
Coin dime = new Coin(0.1, "Dime");
Measurable x = dime;
System.out.println(x.getMeasure());
// Cast esplicito:
Coin theSameDime = (Coin) x;
System.out.println(theSameDime.
                   getDescription());
```

Conversioni di tipo e cast

- In questo caso particolare siamo sicuri che il cast è corretto
- In generale però se si fa un cast verso un certo tipo e l'oggetto in realtà è di un altro tipo non compatibile allora il programma lancerà un'eccezione
- Il cast esplicito tra numeri era sempre possibile a patto di accettare di perdere informazione
- Il cast fra tipi riferimento ad oggetti non è sempre possibile

Conversioni di tipo e cast

- Esiste un operatore per verificare se un certo riferimento punta ad un oggetto di un certo tipo
- L'operatore si chiama **`instanceof`**
- Esempio di uso:

```
if (x instanceof Coin) {
   Coin c = (Coin) x;
   ...
} else ....
```

Costanti nelle interfacce

- Le interfacce non possono contenere variabili istanza
- Però possono contentere costanti
- Nel definire una costante in una interfaccia si possono omettere le parole **`public static final`** perché sono implicite: in quel contesto sono ammesse solo variabili di questo tipo

Costanti nelle interfacce

```
public interface SwingConstants {
   int NORTH = 1;
   int NORT_EAST = 2;
   int EAST = 3;
   ...
}
```

Selezione posticipata

- Quando usiamo una variabile il cui tipo è un'interfaccia sappiamo che in realtà l'oggetto puntato è di una delle classi che implementano l'interfaccia

```
Measurable x = ...;
```

- Quando chiamo il metodo **`getMeasure()`** su **`x`** quale metodo viene invocato?
- La JVM va a vedere il tipo T **vero** dell'oggetto puntato da **`x`** e, in base a questo, esegue il metodo **`getMeasure()`** implementato nella classe T!

Selezione posticipata

- Se la classe T è, ad esempio, **`BankAccount`** viene eseguito il metodo **`getMeasure`** della classe **`BankAccount`**
- Se la classe T è **`Coin`** invece viene eseguito il metodo **`getMeasure`** della classe **`Coin`**
- ...
- Questo principio, secondo cui **il tipo effettivo di un oggetto determina il metodo da chiamare**, è detto **selezione posticipata**

Variabili Polimorfe

- La stessa elaborazione (chiamata ad un metodo di una interfaccia I da un riferimento di tipo I) funziona per oggetti di forme diverse e si adatta alla natura degli oggetti
- La variabile di tipo **`Measurable`** è detta **polimorfa** poiché durante l'esecuzione del programma può riferire oggetti di classi diverse in momenti diversi

Polimorfismo vs Sovraccarico

- Il sovraccarico (overloading) dei nomi si può avere fra diversi metodi di una classe
- Una classe, cioè, può avere diversi metodi con lo stesso nome, ma ognuno deve avere segnatura diversa (per segnatura di un metodo si intende la lista dei tipi dei parametri del metodo stesso)
- Ad esempio per **`BankAccount`** abbiamo definito due costruttori con segnatura diversa

Polimorfismo vs Sovraccarico

`BankAccount()`

`BankAccount(double)`

- Quando un certo nome è sovraccarico è il compilatore che decide quale metodo va effettivamente applicato scegliendo quel metodo la cui segnatura fa match con i tipi dei parametri passati nella chiamata
- Nel caso della chiamata di un metodo polimorfico, invece, il compilatore non può decidere nulla!

Polimorfismo vs Sovraccarico

- Solo al momento dell'effettiva chiamata del metodo durante l'esecuzione si ha a disposizione l'informazione su quale metodo, fra i tanti potenzialmente disponibili, chiamare
- È la JVM che, a runtime, si occupa di andare a guardare il tipo effettivo dell'oggetto e a chiamare quindi il metodo giusto

Interfacce strategiche

- La soluzione che abbiamo adottato per **`DataSet`** usando l'interfaccia **`Measurable`** e il polimorfismo presenta delle limitazioni:
  1. Si può aggiungere l'implementazione dell'interfaccia **`Measurable`** solo a classi che sono sotto il proprio controllo: ad esempio non si può, a meno di non ricompilare tutti i sorgenti dopo averlo fatto, modificare il codice di una classe della libreria standard come **`Rectangle`**

Interfacce strategiche

2. Un oggetto può essere misurato in un unico modo: non si può usare più di una misura. Se si vuole, ad esempio, misurare un conto bancario sia con il saldo che con il tasso di interesse questo non può essere ottenuto con la soluzione che abbiamo visto

- Ripensiamo alla classe **`DataSet`**

Interfacce strategiche

- Un **`DataSet`** deve poter misurare gli oggetti che vi vengono inseriti
- Quando agli oggetti viene chiesto di essere **`Measurable`**, la responsabilità della misurazione è tutta su di loro
- Da questo scaturiscono le limitazioni che abbiamo visto
- Sarebbe comodo se un'entità diversa dagli oggetti si occupasse della loro misurazione

Interfacce strategiche

- Realizziamo questo creando una nuova interfaccia:

```
public interface Measurer {
   double measure(Object anObject);
}
```

- Il metodo **`measure`** misura un oggetto e restituisce tale misurazione
- Usiamo la proprietà, di tutti gli oggetti in Java, di poter essere visti come oggetti della classe **`Object`**

Interfacce strategiche

- Semplicemente basta pensare che **`Object`** rappresenta il minimo comune denominatore di tutti gli oggetti possibili
- La classe **`DataSet`** migliorata richiederà nel costruttore un oggetto **`Measurer`** che userà poi per misurare gli oggetti che verranno aggiunti via via

Interfacce strategiche

```
...
private Measurer measurer;
...
public void add(Object x){
   sum = sum + measurer.measure(x);
   if (count == 0 || measurer.measure(maximum) <
                     measurer.measure(x))
     maximum = x;
   count++;
}
...
```

Interfacce strategiche

- A questo punto possiamo definire tutti i **`Measurer`** che vogliamo per qualsiasi classe
- Ad esempio prendiamo **`Rectangle`**
- Vogliamo definire un misuratore per oggetti **`Rectangle`** che usi l'area come misura
- Basta creare una nuova classe che implementi **`Measurer`**
- Questa classe non ha bisogno di nessuna variabile istanza e dovrà semplicemente implementare un metodo **`measure`**

Interfacce strategiche

```
class RectangleMeasurer implements
                                Measurer {
public double measure(Object anObject) {
  Rectangle aRectangle = (Rectangle)
                                anObject;
  double area = aRectangle.getWidth() *
                aRectangle.getHeight();
  return area;
}
}
```

Interfacce strategiche

- **`RectangleMeasure`** è una classe che ha un compito molto preciso e circoscritto
- Serve a misurare esclusivamente oggetti della classe **`Rectangle`** in base all'area e deve venire passato al costruttore di un oggetto **`DataSet`** che sarà così in grado di misurare oggetti della classe **`Rectangle`**
- Il metodo measure deve rispettare la segnatura imposta dall'interfaccia quindi dovrà avere un parametro di tipo **`Object`**

Interfacce strategiche

- L'implementazione di measure della classe **`RectangleMeasurer`** fa un cast esplicito a **`Rectangle`**
- Se l'oggetto che viene misurato non è un **`Rectangle`** sarà sollevata un'eccezione (perdiamo la possibilità di avere **`DataSet`** a cui vengono aggiunti oggetti di diverso tipo)
- Interfacce come **`Measurer`** vengono dette **strategiche** poiché gli oggetti delle classi che le implementano mettono in atto una particolare strategia di elaborazione

Interfacce strategiche

- Ad esempio **`RectangleMeasure`** ha come strategia quella di misurare i **`Rectangle`** in base all'area
- Per realizzare una diversa strategia si usa semplicemente un oggetto strategico diverso
- Ad esempio potremmo definire un'altra classe **`PerimeterRectangleMeasurer`** che utilizza il perimetro come misura

Interfacce strategiche

- Questo tipo di soluzione si trova spesso nelle implementazioni delle classi che realizzano l'interfaccia grafica e la gestione degli eventi in Java

Classi interne

- Classi come **`RectangleMeasurer`** hanno uno scopo limitato e possono tranquillamente venire definite come interne ad altre classi o a metodi
- Nel seguente esempio la classe **`RectangleMeasurer`** viene definita dentro un metodo **`main`** al solo scopo di crearne un oggetto da passare a **`DataSet`**

Classi interne

```
...
public static void main (String[] args) {
   class RectangleMeasurer implements
                              Measurer {
      ...
    }
Measurer m = new RectangleMeasurer();
DataSet data = new DataSet(m);
...
```

Test

- Consultare il codice allegato **`DataSetStrat.java`** e **`DataSetStratTest.java`** per l'implementazione di **`DataSet`** con l'interfaccia strategica **`Measurer`**

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);
```

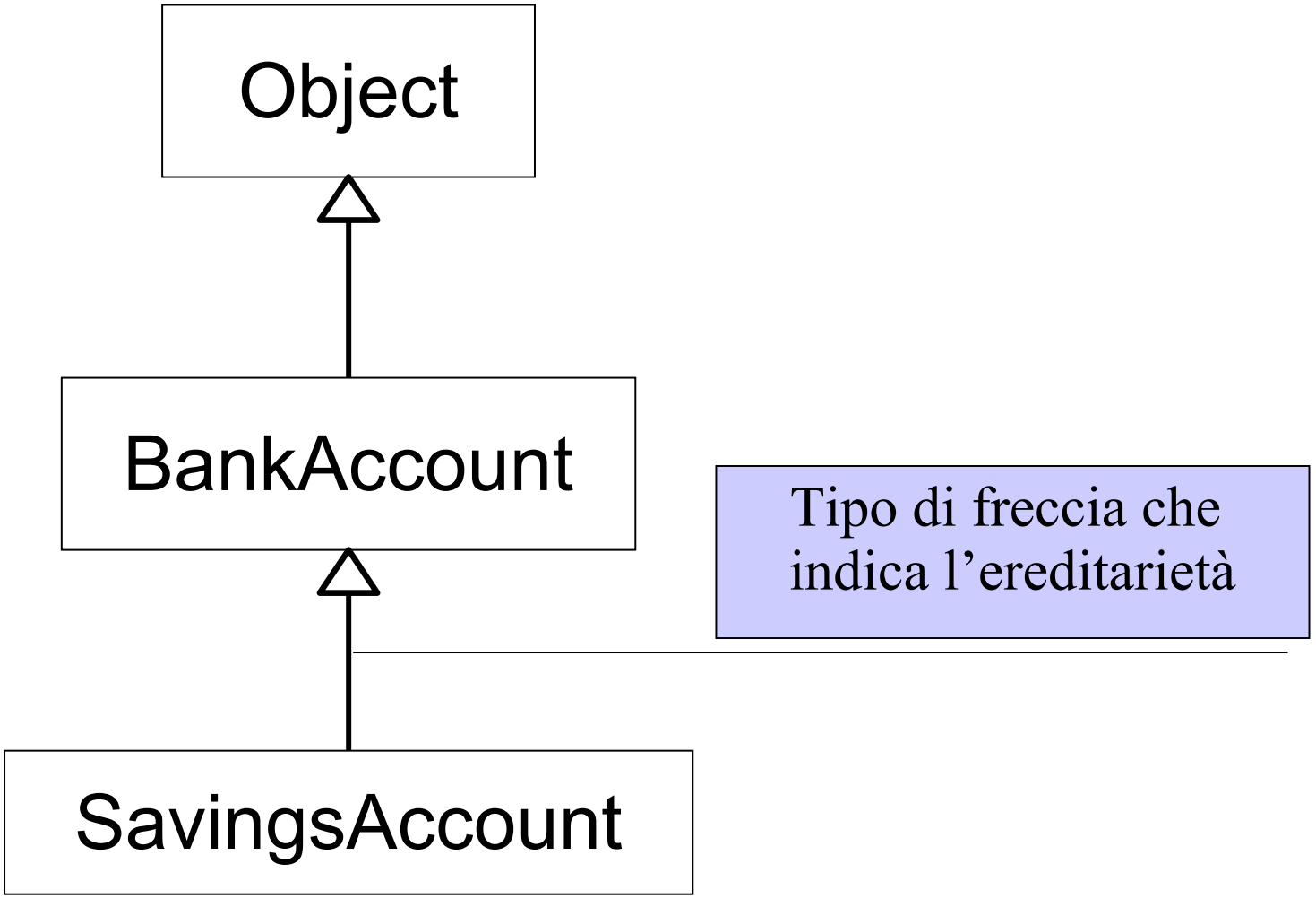
Ereditarietà: terminologia

- 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

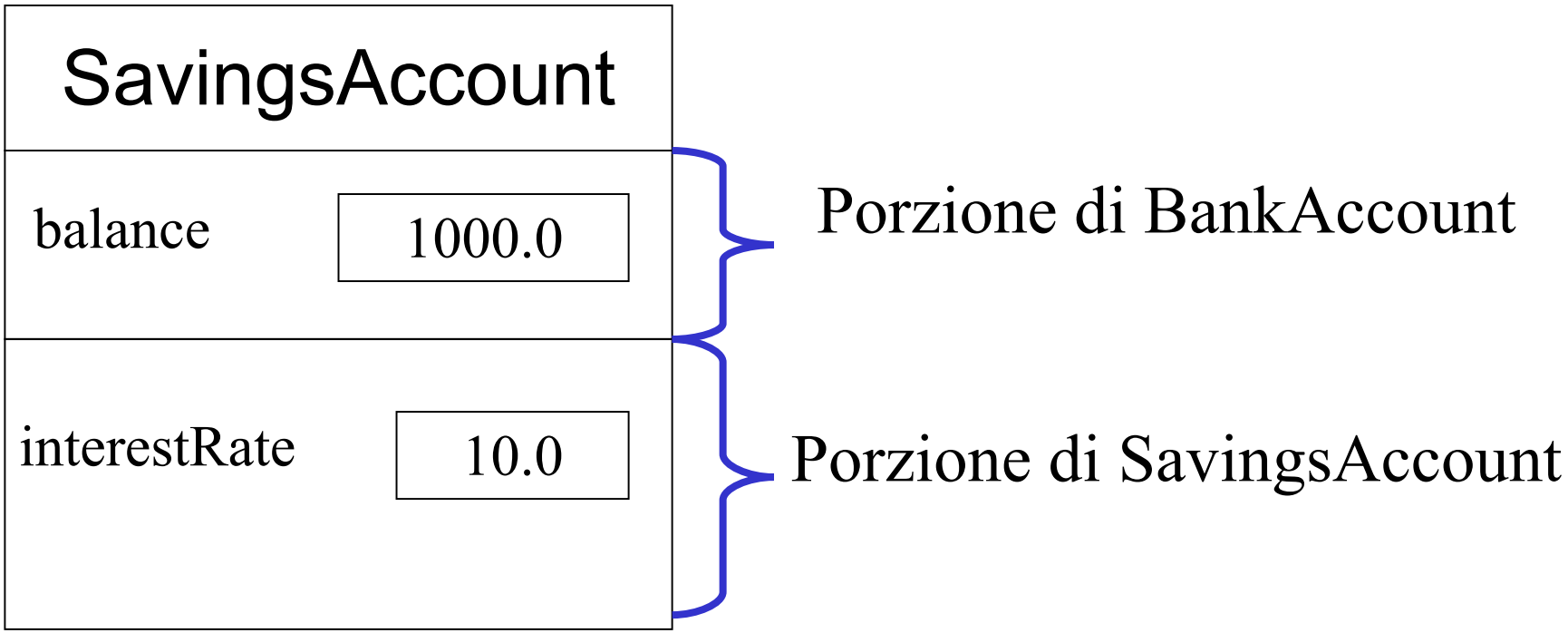
- La sottoclasse può contenere:
  - Nuovi metodi
  - Nuove variabili istanza
  - Metodi della superclasse sovrascritti
- Nell'esempio abbiamo che in **`SavingsAccount`** dobbiamo aggiungere una variabile istanza per memorizzare il tasso di interesse e un nuovo metodo che aggiunge gli interessi maturati

SavingsAccount

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

}
```

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)`

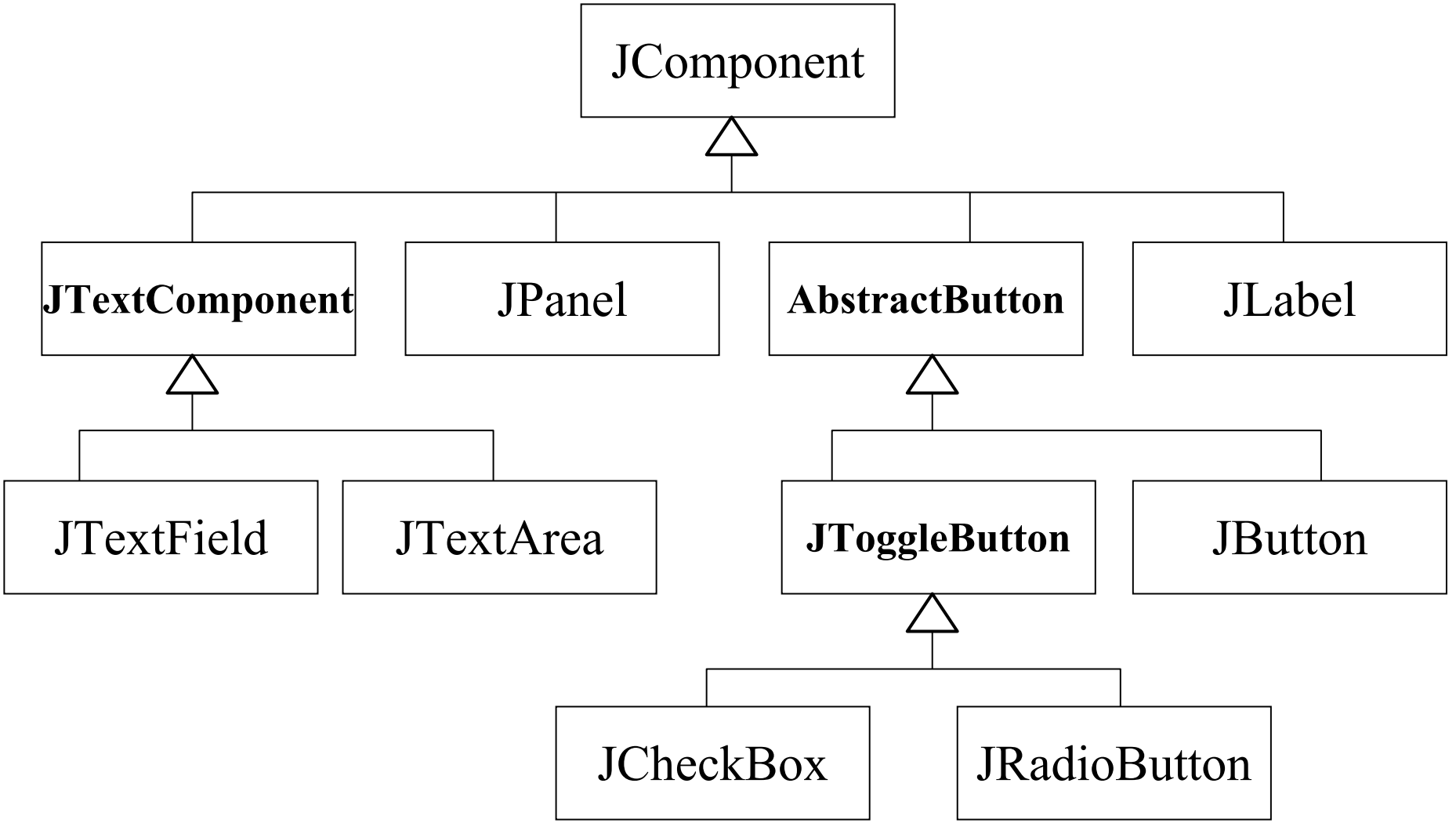
Giustificazione del nome

- 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

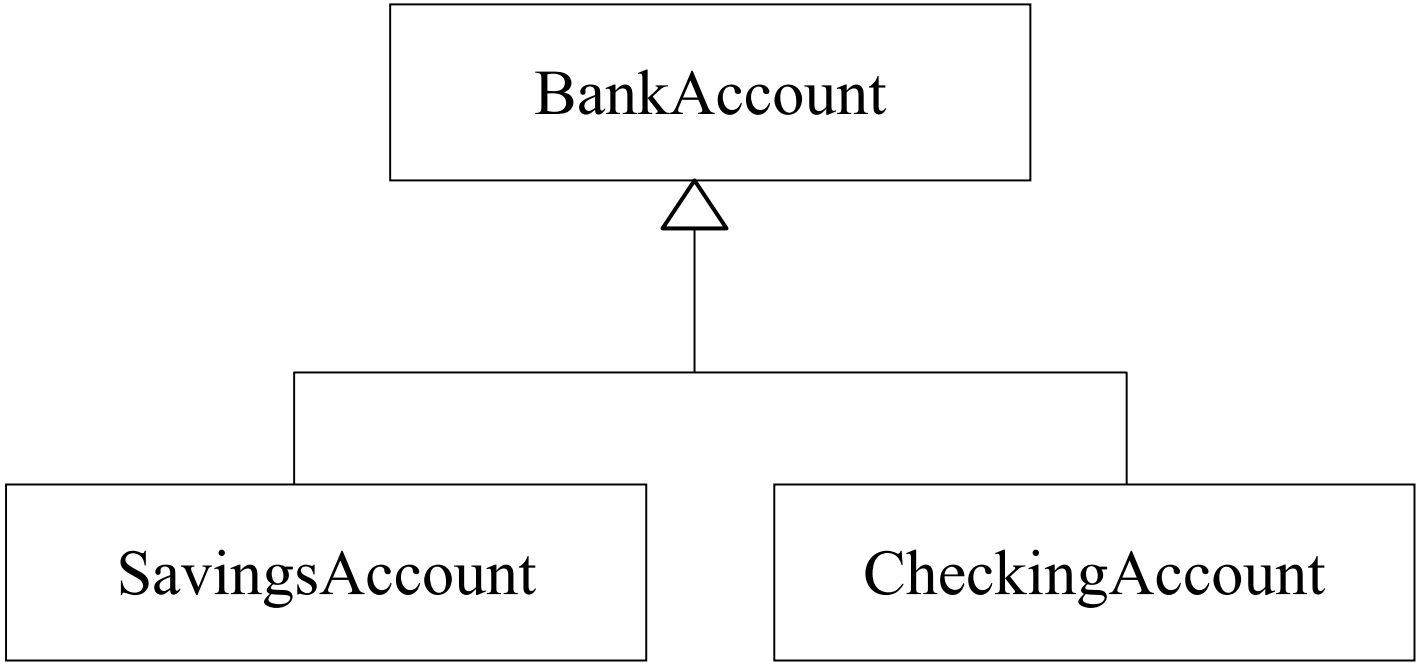
Esempio: gerarchia di oggetti grafici Swing



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

1. 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

2. 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

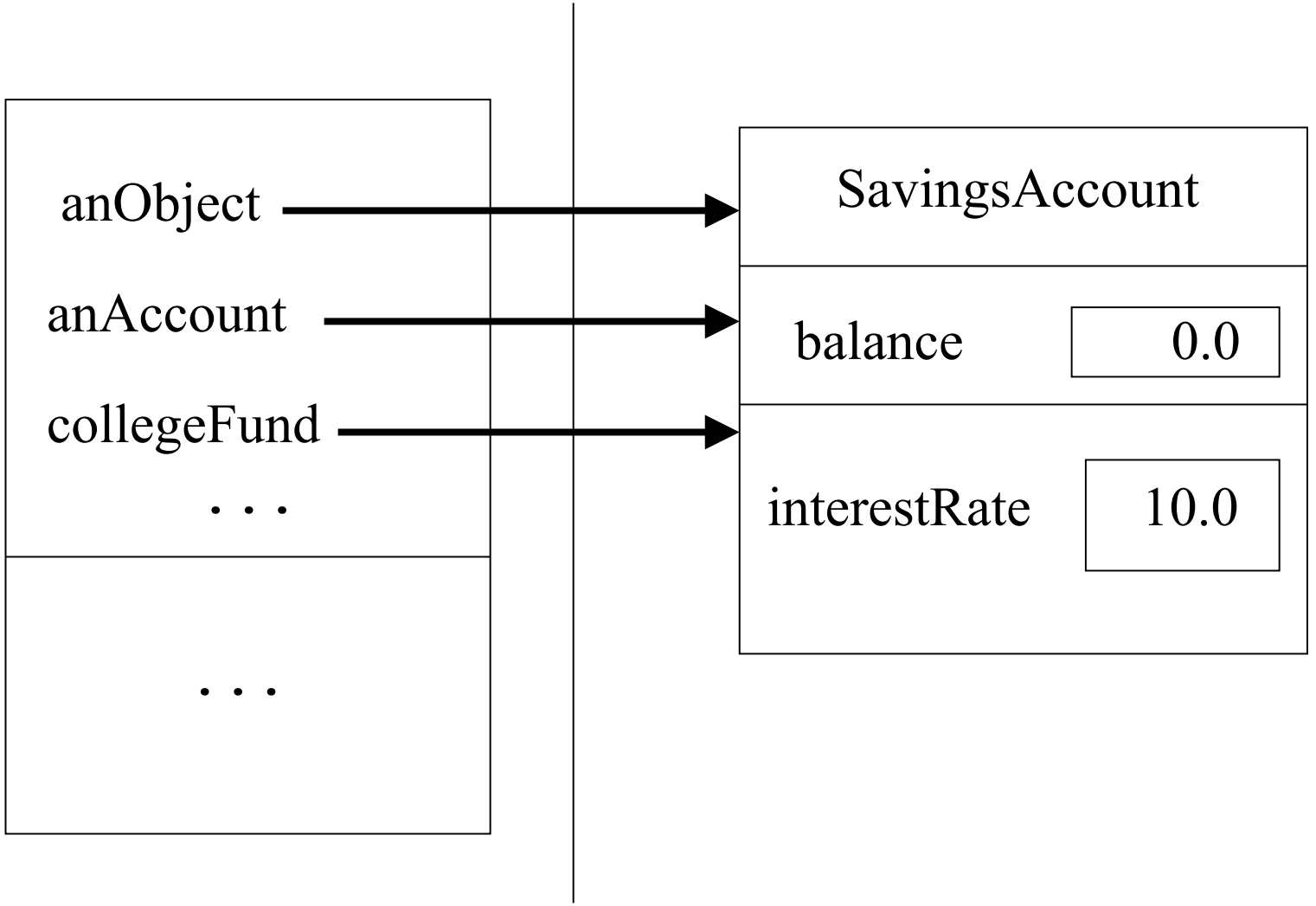
```
public class CheckingAccount extends
                                    BankAccount {
  public CheckingAccount(double
                         initialBalance) {
    // Costruisce la superclasse
    super(initialBalance);
    // Inizializza il contatore di
    // operazioni
    transactionCount =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

- 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) {
...
} }
```