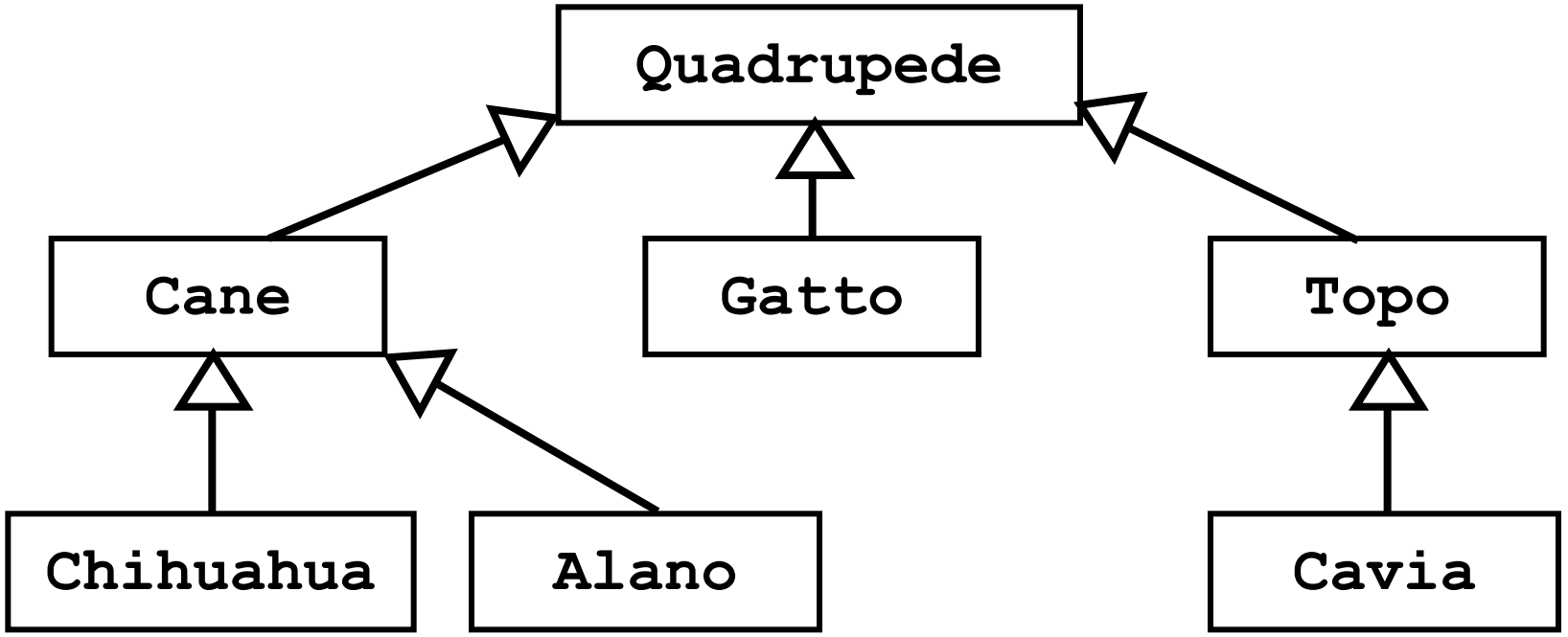
Ereditarietà

- Per definire stato e comportamento di nuovi oggetti, è utile avere una base da cui partire
- In particolare, un nuovo oggetto potrebbe essere un **caso particolare** di una tipologia di oggetti esistente, già definita in precedenza.
- Java mette a disposizione per questo scopo un meccanismo di estensione delle classi.
  - Una classe, detta **superclasse**, può essere specializzata definendo una **sottoclasse** che ne contenga casi particolari.

Ereditarietà

- Una superclasse e una sua sottoclasse sono legate da un meccanismo di ereditarietà.
- La sottoclasse **eredita** da una classe base (superclasse) lo stato e il comportamento.
- Questo significa che la sottoclasse possiede tutti i campi e metodi della superclasse.
- Però nella sottoclasse si possono aggiungere o modificare campi e metodi per **specializzare** la superclasse.

Ereditarietà



Ereditarietà

- L'ereditarietà può essere vista:
  - come una semplificazione logica: gli oggetti della sottoclasse possono eseguire tutte le operazioni previste dalla superclasse
  - come una forma di riutilizzo del codice: non c'è bisogno di ri-descrivere quanto viene ereditato
- Viceversa, le modifiche apportate alla sottoclasse sono limitate solo ad essa e non si applicano alla superclasse.

Ereditarietà

- I termini “sottoclasse” e “superclasse” derivano dalla teoria degli insiemi (classe = insieme).
- Ma in effetti un oggetto di una sottoclasse è più grande (più potente, più specializzato) di un oggetto della superclasse.
- Per un’immagine più chiara può essere più intuitivo pensare che la sottoclasse estende (in inglese: **extends**) la superclasse, e parlare rispettivamente di classe estesa e classe base.

Ereditarietà multipla

- Alcuni linguaggi di programmazione permettono che una classe possa ereditare metodi e caratteristiche da più classi base.
  - Tuttavia questo può creare inconvenienti: se due classi hanno una stessa caratteristica, quale viene ereditata dalla classe estesa?
- Per evitare problemi, Java usa un modello a ereditarietà singola. Una sorta di ereditarietà multipla viene però introdotta dalle **interfacce**.

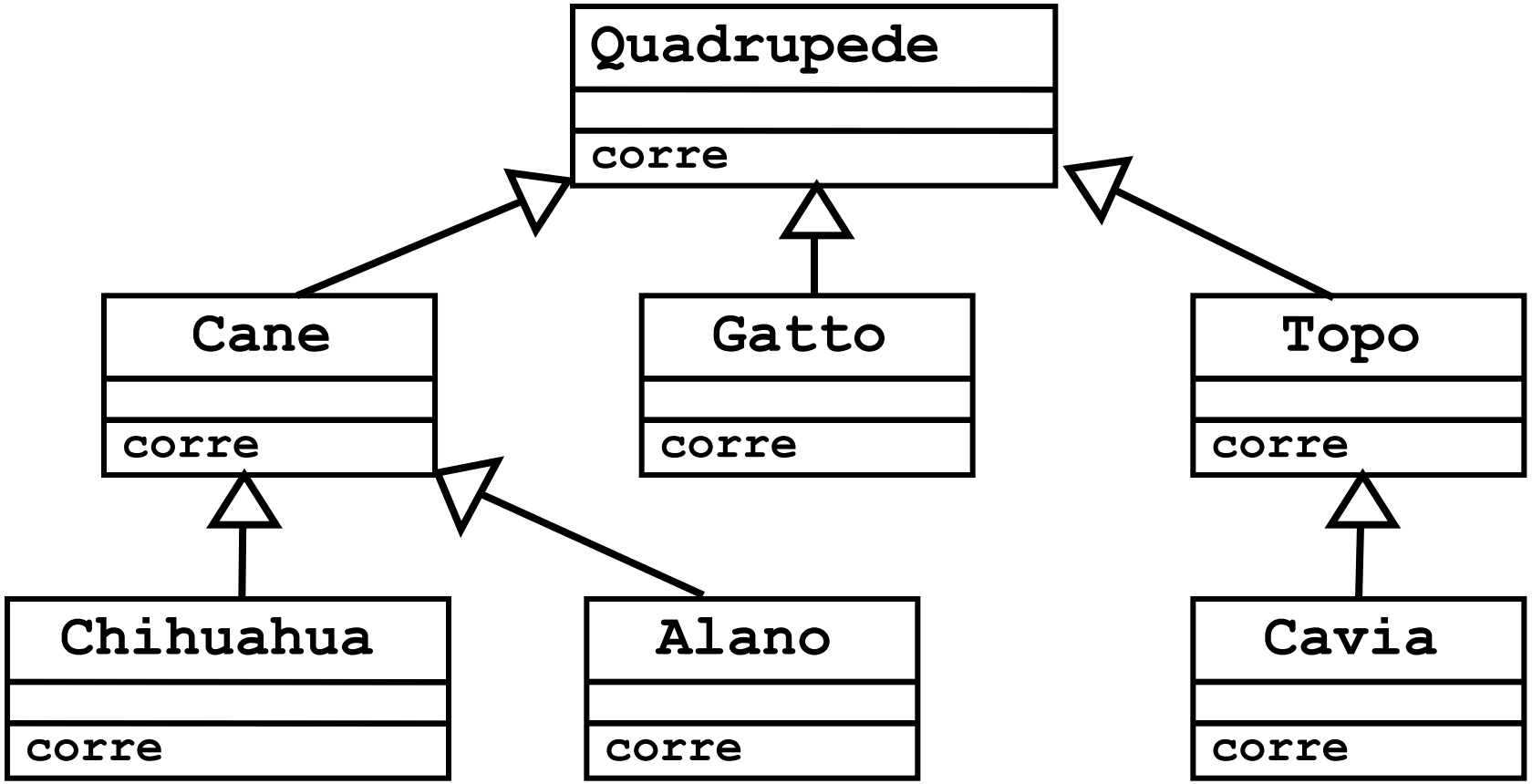
Ereditarietà

- In Java, una sottoclasse eredita da una sola classe base.
- Quindi è bene prevedere definizioni di classi generiche, e poi specializzarle in sottoclassi.
- Un’istanza di una sottoclasse è anche effettivamente istanza della superclasse, quindi ne può ereditare e usare i metodi.

Polimorfismo

- Una classe estesa eredita le caratteristiche della classe di base che essa estende.
- I metodi della classe base sono automaticamente definiti per la classe estesa.
- In aggiunta la classe estesa può:
  - definire altri metodi non previsti
  - ridefinire i metodi della classe base
- Per uno stesso nome di operazione si possono avere implementazioni diverse: **polimorfismo**.

Polimorfismo



- Grazie al polimorfismo, un metodo **`corre()`** può essere “adattato” alle diverse classi.

Ereditarietà in Java

- Consideriamo una classe **BankAccount**, che modella un conto di risparmio.

```
class BankAccount
{ //campi
  private double balance;
  //costruttori
  public BankAccount(double x) { balance = x; }
  public BankAccount() { this(0); }
  //metodi
  public void deposit(double x) { balance += x; }
  public void withdraw(double x) { balance -= x; }
  public double getBalance() { return balance; }
  public void transfer(BankAccount b, double x)
  { this.withdraw(x); b.deposit(x); }
}
```

Ereditarietà in Java

- Con questa classe, è lecito scrivere:

```
contoDiGianni = new BankAccount();
risparmiDiNonna = new BankAccount(4283.15);
contoDiGianni.deposit(1000);
contoDiGianni.withdraw(250);
risparmiDiNonna.transfer(contoDiGianni,500);
double x = contoDiGianni.getBalance();
```

- Per esercizio, provare a migliorare la classe con:
  - vietare prelievi superiori a 1000 e/o superiori al saldo
  - aggiungere un campo **`String filiale`** e un metodo con un parametro **`String`** che applica una commissione di 2.50 euro se è diverso da **`filiale`**

Ereditarietà in Java

- Vogliamo definire una classe **`SavingsAccount`** che modella un conto di interesse.
- Un conto di interesse è un **caso speciale** di libretto di risparmio. Quindi, ha senso definire **`SavingsAccount`** come sottoclasse della classe **`BankAccount`** già definita.
- Per farlo si usa la parola chiave **`extends`**:

```
class SavingsAccount extends BankAccount
```

Ereditarietà in Java

- Nella definizione di `SavingsAccount` vanno messi solo i campi e i metodi nuovi.
- Ad esempio, la classe `SavingsAccount` avrà una variabile istanza in più, per descrivere il tasso di interesse, e un metodo in più, per aggiungere gli interessi maturati a fine anno.
- Però non c'è bisogno di ridefinire il metodo **`deposit`**, che è ereditato dalla superclasse `BankAccount` e funziona in modo identico.

Ereditarietà in Java

- Un oggetto della classe `SavingsAccount` è anche un oggetto della classe `BankAccount` che però sa fare qualcosa di più.
- Quindi eredita dalla superclasse variabili e metodi (vedremo come).
- Però in Java i costruttori **non sono ereditati**.
  - Un'istanza della sottoclasse avrà più campi e sarà più complessa da inizializzare.
  - Un costruttore ritorna un riferimento di tipo superclasse, ma servirebbe di tipo sottoclasse.

Ereditarietà in Java

- Definizione della classe **SavingsAccount**:

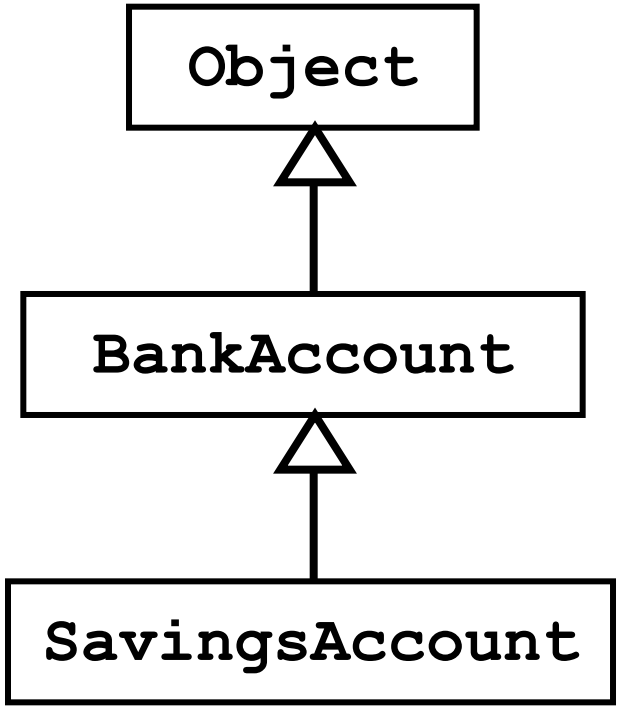
```
class SavingsAccount extends BankAccount
{ //campi
  private double interest;
  //costruttori
  public SavingsAccount(double r)
  { interest = r; }
  //metodi
  public void addInterest()
  { double y = getBalance() * interest / 100;
    deposit(y); // vale this.deposit(y)
  }
}
```

Ereditarietà in Java

- Definendo quindi un oggetto **`contoDiAnna`** della classe **`SavingsAccount`**, anche su di esso è possibile usare i metodi **`deposit()`**, **`withdraw()`**, **`getBalance()`**.
- È pure possibile usarlo come parametro in un metodo che richiede un oggetto **`BankAccount`** come ad esempio **`transfer()`**.
- Addirittura, **`addInterest()`** invoca a sua volta **`getBalance()`** e **`deposit()`** implicitamente sull'oggetto **`this`** (di tipo **`SavingsAccount`**).

Ereditarietà in Java

- In Java, tutte le classi si considerano sottoclassi di una speciale classe **`Object`**.
- Ecco perché abbiamo potuto introdurre metodi di utilità generali: sono i metodi della classe **`Object`** che sono automaticamente ereditati da ogni classe.



Metodi della classe `Object`

- `protected Object clone()`:
  copia l'oggetto e ritorna un riferimento ad esso
- `boolean equals(Object obj)`:
  verifica se "l'altro oggetto" (`obj`) è uguale a questo
- `protected void finalize()`:
  viene eseguito dal garbage collector prima di liberare la memoria occupata dall'oggetto
- `String toString()`:
  restituisce una rappresentazione stringa dell'oggetto

Overriding

- Quando estendiamo una classe, nella sottoclasse vanno definiti:
  - nuovi campi, se esistono, oltre a quelli ereditati.
  - nuovi metodi, se esistono, oltre a quelli ereditati.
  - nuovi costruttori. I costruttori sono sempre nuovi, non si ereditano mai, anche se esistono “scorciatoie” per sfruttare i costruttori della superclasse.

Overriding

- Java consente anche di **riscrivere** i metodi esistenti nella superclasse (overriding).
- L'overriding deve rispettare i tipi coinvolti, e perciò vale solo per i metodi. Non avrebbe senso per costruttori e variabili di istanza.
  - In **`SavingsAccount`** non è corretto dichiarare una nuova variabile **`balance`**. Si può invece dichiarare un nuovo metodo **`withdraw()`** con gli stessi tipi di quello della superclasse.
- Non si può fare overriding di metodi statici.

Overriding vs. overloading

OVERLOADING

- stesso nome
- diverso numero e tipo di parametri (firma)
- diverso tipo di ritorno (se la firma è diversa)
- due metodi che coesistono fra loro

OVERRIDING

- stesso nome
- stesso numero e tipo di parametri (firma)
- stesso tipo di ritorno (stesso prototipo)
- due metodi di cui uno rimpiazza l’altro

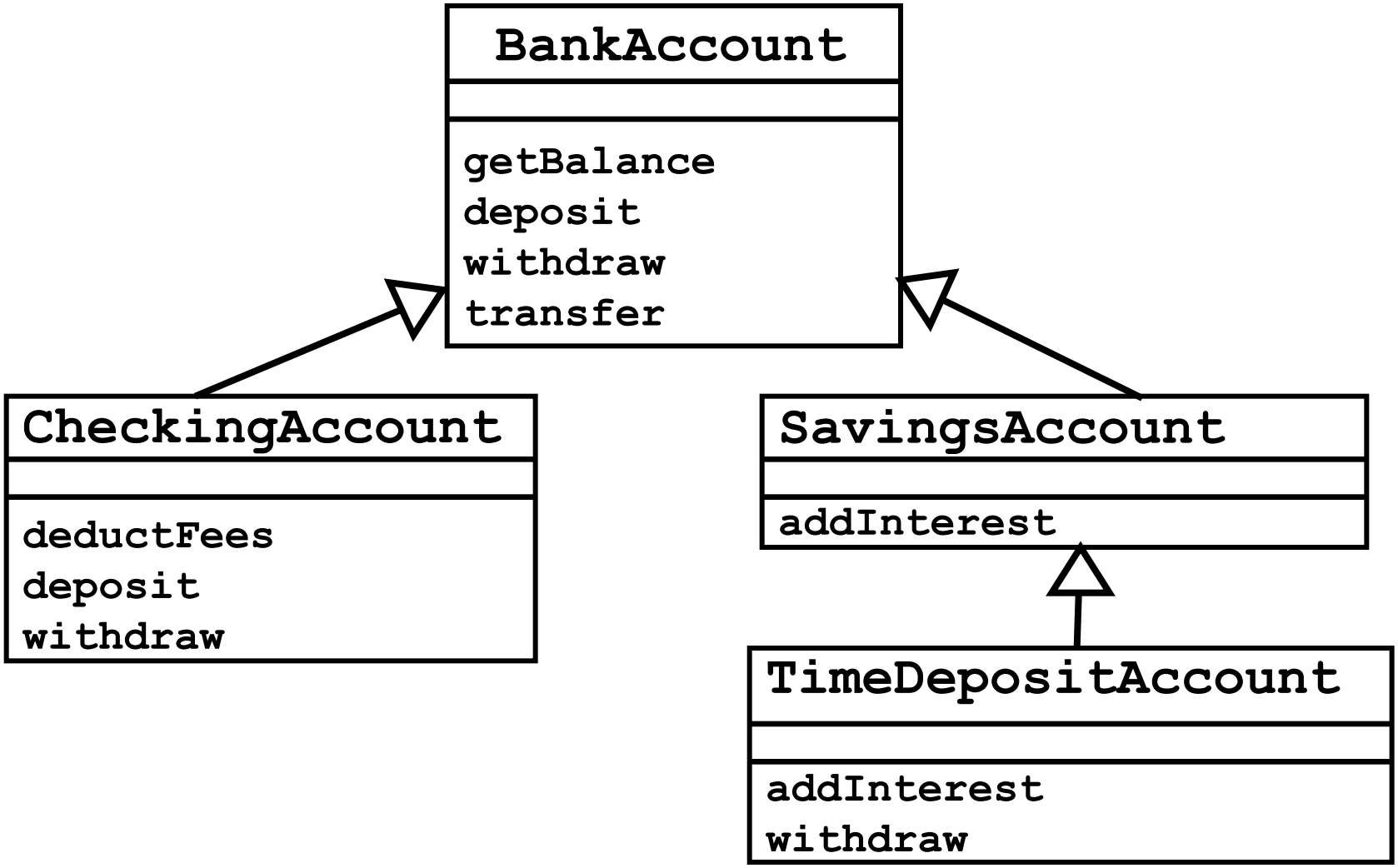
Overriding e interfaccia

- Per estendere una superclasse basta conoscere la sua interfaccia, non è necessario sapere come sono implementati i metodi.
- Questo consente di prescindere da modifiche all'implementazione della superclasse.
- Se ci piace il metodo della superclasse, lo lasciamo così; altrimenti lo sovrascriviamo.
- Solitamente è bene fare overriding dei metodi ereditati automaticamente dalla classe **`Object`**.

Progettare una sottoclasse

- Vogliamo estendere la classe **`BankAccount`**, comprendendo **`SavingsAccount`** e altre due classi: **`CheckingAccount`** (conto corrente) e **`TimeDepositAccount`** (conto vincolato).
- Il conto corrente è come un libretto di risparmio ma ha un costo per ogni operazione. Le prime 10 sono gratis, poi si paga 1.50 .
- Il conto vincolato è come un conto di interesse, ma ha una penalità se si ritirano i soldi prima di fine anno.

Progettare una sottoclasse



Progettare una sottoclasse

- Abbiamo identificato **`CheckingAccount`** nella struttura gerarchica delle classi. Ora dobbiamo:
  - inserire una nuova variabile d'istanza chiamata **`transactions`** (conta le operazioni effettuate)
  - inserire due nuove variabili di classe costanti (il costo delle operazioni e il numero di quelle gratis)
  - scrivere un nuovo metodo **`deductFees()`**
  - riscrivere i metodi **`deposit()`** e **`withdraw()`** (devono incrementare il numero di operazioni)

La parola chiave **super**

- Consideriamo il metodo **deposit()**

```
class CheckingAccount extends BankAccount
{ //campi
  private static final int FREE_TRANSACTIONS = 10;
  private static final double TR_FEE = 1.50;
  private int transactions;
  //costruttori
  ...
  //metodi
  public void deposit(double x)
  { transactions++;
    ??? aumenta il bilancio di x;
  }
}
```

se mettiamo
balance += x;
non funziona!
balance è privata

La parola chiave `super`

- Non dovrebbe esserci problema: i campi sono privati, ma hanno **metodi d'accesso pubblici**.
  - Qua verrebbe da usare **`deposit(x)`**

```
class CheckingAccount extends BankAccount
{  ...
   ...
   //metodi
   public void deposit(double x)
   {  transactions++;
      deposit(x);
   }
}
```

questo vale come **`this.deposit(x);`**
ma allora non va bene!
è questo stesso metodo
(ricorsione infinita)

La parola chiave **super**

- Vogliamo chiamare sì il metodo **deposit()**, ma quello della superclasse. Per tale scopo esiste la parola chiave **super**.

```
class CheckingAccount extends BankAccount
{  ...
   ...
   //metodi
   public void deposit(double x)
   {  transactions++;
      super.deposit(x);
   }
}
```

adesso l'istruzione può essere correttamente compilata ed eseguita

La parola chiave **`super`**

- La parola chiave **`super`** è in effetti analoga alla parola chiave **`this`**. Come quest'ultima può essere usata in due contesti differenti:
  - per identificare un membro della superclasse (vedi **`super.deposit()`** vs. **`this.deposit()`**)
  - per identificare il costruttore della superclasse
- In altre parole, **`super`** è un riferimento universale a quanto precede nella gerarchia dell'ereditarietà.

La parola chiave **`super`**

- Anche se i costruttori non sono ereditati, possono essere semplificati usando **`super`**.
  - Se vogliamo che ogni **`CheckingAccount`** sia costruito con il bilancio iniziale (parametro) e il conto delle operazioni eseguite pari a zero...

```
class CheckingAccount extends BankAccount
{  ...
   //costruttori
   public CheckingAccount(double x)
   {  super(x);
      transactions = 0;
   }
}
```

Costruttori della superclasse

- Abbiamo già visto che un costruttore può invocarne un altro chiamando **`this()`**. Questa deve essere la prima istruzione eseguita dal costruttore.
- Invece di usare **`this()`** si potrebbe usare analogamente **`super()`**. Questo ha senso: il costruttore per prima cosa si preoccupa di costruire la parte ereditata dalla superclasse.
- Se usa un'invocazione al costruttore **`super()`**, il costruttore della classe estesa lo deve fare per prima cosa (cioè come **prima istruzione**).

Costruttori della superclasse

- Quindi un costruttore può invocare **`this()`** o **`super()`** come prima istruzione.
- Se non lo fa, rimane comunque il problema di costruire la parte ereditata dalla superclasse.
- Quindi se anche l'uso di **`super()`** viene omesso si usa il costruttore predefinito della superclasse (quello senza parametri).
- Nell'esempio di **`SavingsAccount`** facevamo implicitamente leva su questo fatto, visto che **`BankAccount`** di default metteva **`balance`** a 0.

Costruttori della superclasse

- Se il costruttore di una classe estesa non invoca subito un **`super()`** viene aggiunto di default un **`super()`** senza parametri.
  - Dato che **`BankAccount()`** esiste, è lecito:

```
class CheckingAccount extends BankAccount
{  ...
   //costruttori
   public CheckingAccount(double x)
   { //senza dir niente, viene invocato BankAccount()
     super.deposit(x);
     transactions = 0;
   }
}
```

Costruttori della superclasse

- Ma se il costruttore della superclasse senza parametri manca: errore in compilazione.
  - Nell'esempio di prima, **`TimeDepositAccount`** è una sottoclasse di **`SavingsAccount`**, priva di costruttore di default. Il costruttore della superclasse va perciò invocato esplicitamente.

```
class TimeDepositAccount extends SavingsAccount
{ ...
  //costruttori
  public TimeDepositAccount(double r)
  { super(r);
    ...
  }
}
```

Costruttori

- Riassumendo, valgono le regole:
  - se non esiste un costruttore, Java ne crea uno di default che non ha parametri e non fa niente; se ne esiste almeno uno, bisogna usare quello
  - se il costruttore di una sottoclasse non incomincia con **`this()`** o con **`super()`**, viene invocato il costruttore di default della superclasse
- Queste regole non valgono però per la classe **`Object`** che non invoca mai il costruttore della superclasse perché non ha superclasse.

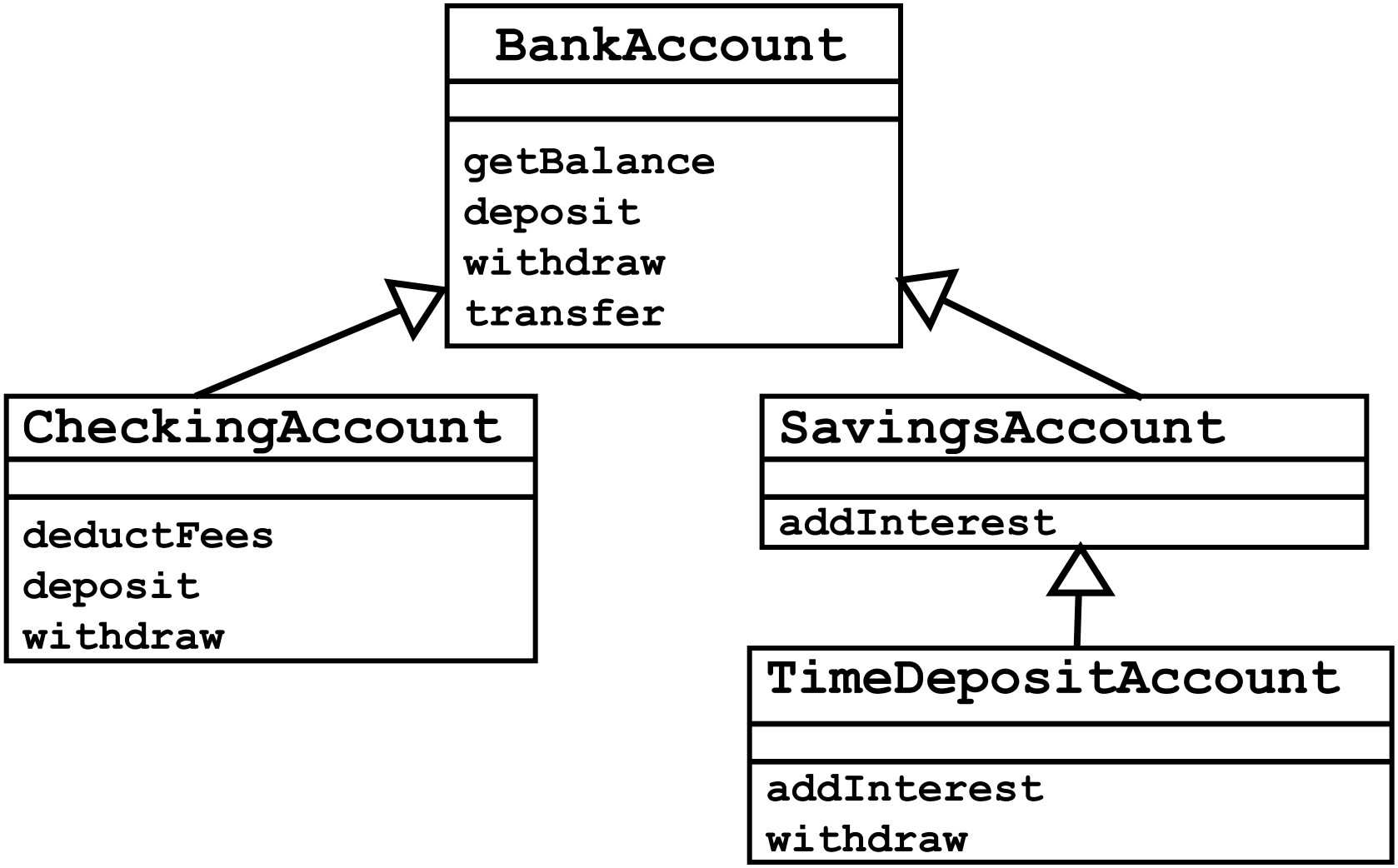
Metodi ed ereditarietà

- I metodi ridefiniti nelle sottoclassi hanno la precedenza sui metodi delle superclassi.
- Quando un metodo è sotto overriding, la versione dominata non viene affatto eseguita, a meno che non sia invocata tramite **`super`**.
- Per i metodi di cui non si fa overriding invece, se invocati nelle sottoclassi si sale nella gerarchia fino alla superclasse di definizione

Metodi ed ereditarietà

- Se in un metodo o un costruttore di una sottoclasse si invoca:
  - `this.metodo()`: viene eseguito il `metodo()` della sottoclasse stessa
  - `super.metodo()`: viene eseguito il `metodo()` della superclasse (il quale può essere a sua volta ereditato da una superclasse)
  - `metodo()`: è sottinteso `this`, quindi viene eseguito il `metodo()` della sottoclasse stessa

Riprendiamo l'esempio..



Metodi ed ereditarietà

- Se non c'è overriding, l'ereditarietà si propaga
  - La classe **`CheckingAccount`** ridefinisce il metodo **`deposit()`**. Ogni sua istanza userà il metodo **`deposit()`** ridefinito, non quello della classe base **`BankAccount`**.
  - La classe **`CheckingAccount`** non ridefinisce il metodo **`getBalance()`**. Ogni sua istanza userà il metodo **`getBalance()`** originario della classe base **`BankAccount`**.

Metodi ed ereditarietà

- Esempi di overriding ed ereditarietà:
  - La classe **`TimeDepositAccount`** non ridefinisce il metodo **`deposit()`**. Ogni sua istanza userà il metodo **`deposit()`** di **`BankAccount`**.
  - Ma in realtà **`BankAccount`** è la superclasse della superclasse; questo perché neanche la prima superclasse che si incontra per prima, **`SavingsAccount`**, fa override di questo metodo.