Lezione

- Ereditarietà ….CONTINUA

Esempio: il conto bancario


```
public class BankAccount
{  private double balance;
   public BankAccount() {  balance = 0;}
   public BankAccount(double initialBalance)
   {   balance = initialBalance;}
   public void deposit(double amount)
   {   balance = balance + amount;}
   public void withdraw(double amount)
   {   balance = balance - amount; }
   public double getBalance(){return balance;}
   public void transfer(BankAccount other,
                        double amount)
   {   withdraw(amount);
       other.deposit(amount);}

}
```

Estendere il codice

- Supponiamo di voler implementare altri tipi di conto bancario.
- Per esempio, consideriamo un conto bancario che dia interessi ad un certo tasso fisso *interestRate*.
- Dobbiamo riscrivere tutto il codice?
- Possiamo sfruttare il codice esistente estendendolo opportunamente?

La classe SavingsAccount


```
public class SavingsAccount extends BankAccount
{ private double interestRate;

   public SavingsAccount(double rate)
   {  interestRate = rate;
   }

   public void addInterest()
   {  double interest = getBalance() * interestRate / 100;
      deposit(interest);
   }

}
```

Sovrascrivere un metodo: Esempio

- Vogliamo modificare la classe **SavingsAccount** in modo che ogni operazione di versamento abbia un costo (fisso) **FEE**, che viene automaticamente addebitato sul conto

```
public class SavingsAccount extends BankAccount
{  ...
   private final static double FEE = 2.58; // euro
}
```

- I versamenti nei conti di tipo **SavingsAccount** si fanno però invocando il metodo **deposit** di **BankAccount**, sul quale non abbiamo controllo

Controllo statico dei tipi

- Java, come molti altri linguaggi, effettua un controllo dei tipi (type checking) statico.
- Statico: fatto dal compilatore prima di iniziare l'esecuzione del programma.
- Dinamico: fatto dall'interprete durante l'esecuzione (*a runtime*)
- Il type checking statico garantisce che non ci saranno errori durante l'esecuzione.

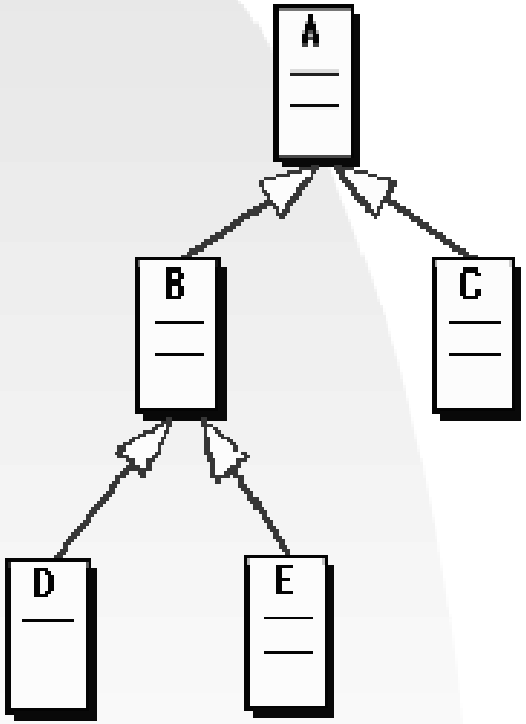
Controllo statico dei tipi

- Type checking statico: il compilatore controlla che per una variabile si chiami un metodo definito per la classe di quella variabile.

```
SavingsAccount s;
BankAccount b;
…
b.getBalance();
s. getBalance();
s.addInterest();
b.addInterest();
```

Conversione fra riferimenti

Polimorfismo




```
A a; B b; D d;
d = new D();
b = new D();
a = new D();
a = b;
```

questi assegnamenti sono tutti legali perché un oggetto di tipo D ha anche tipo B e A

Conversione fra riferimenti

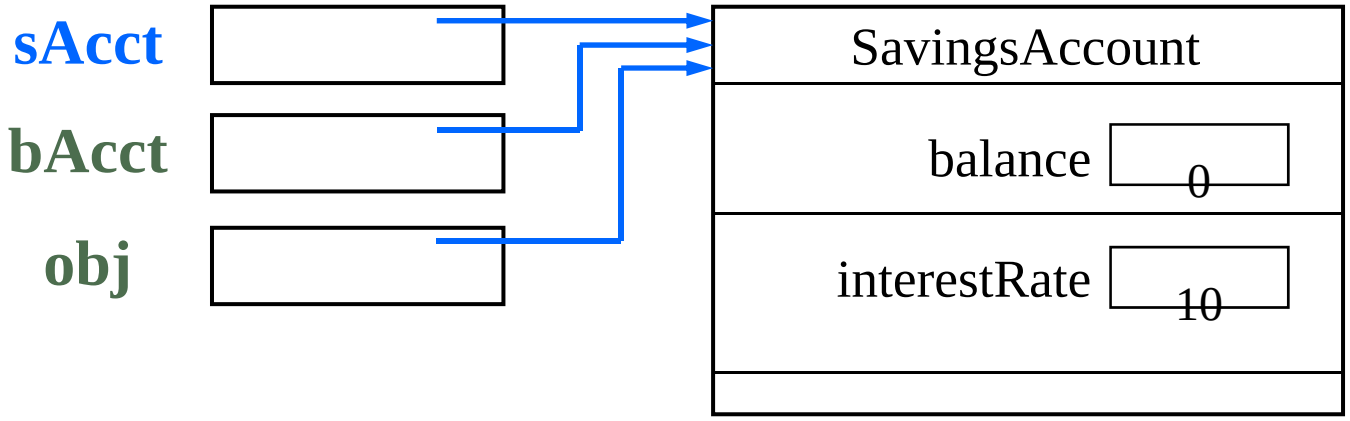
- Un oggetto di tipo **SavingsAccount** è un ***caso speciale*** di oggetti di tipo **BankAccount**
- Questa proprietà si riflette in una proprietà sintattica
  - ***un riferimento ad un oggetto di una classe derivata può essere assegnato ad una variabile oggetto del tipo di una sua superclasse***

```
SavingsAccount sAcct = new SavingsAccount(10);
BankAccount bAcct = sAcct;
Object obj = sAcct;
```

Conversione fra riferimenti

```
SavingsAccount sAcct = new SavingsAccount(10);
BankAccount bAcct = sAcct;
Object obj = sAcct;
```

- Le tre variabili puntano ora ***allo stesso oggetto***



Conversione fra riferimenti

```
SavingsAccount sAcct = new SavingsAccount(10);
BankAccount bAcct = sAcct;
bAcct.deposit(500); // OK
```

- Tramite la variabile **bAcct** si può usare l'oggetto come se fosse di tipo **BankAccount**, senza però poter accedere alle proprietà specifiche di **SavingsAccount**

```
bAcct.addInterest();
```

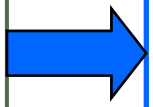
```
cannot resolve symbol
symbol  : method addInterest()
location: class BankAccount
      bAcct.addInterest();
            ^
1 error
```

BankAccount: transferTo

- Aggiungiamo un metodo a **BankAccount**, che useremo negli esempi seguenti

```
public class BankAccount
{  ...
   public void transferTo(BankAccount other,
                          double amount)
   {  withdraw(amount); // this.withdraw(...)
      other.deposit(amount);
   }
}
```

```
BankAccount acct1 = new BankAccount(500);
BankAccount acct2 = new BankAccount();
acct1.transferTo(acct2, 200);
System.out.println(acct1.getBalance());
System.out.println(acct2.getBalance());
```



```
300
200
```

Conversione fra riferimenti

- Quale scopo può avere questo tipo di conversione?
  - per quale motivo vogliamo trattare un oggetto di tipo **SavingsAccount** come se fosse di tipo **BankAccount**?

```
BankAccount other = new BankAccount(1000);
SavingsAccount sAcct = new SavingsAccount(10);
BankAccount bAcct = sAcct;
other.transferTo(bAcct, 500);
```

- Il metodo **transferTo** di **BankAccount** richiede un parametro di tipo **BankAccount** e non conosce (né usa) i metodi specifici di **SavingsAccount**, ma è comodo poterlo usare senza doverlo modificare!

Conversione fra riferimenti

- La conversione tra ***riferimento a sottoclasse*** e ***riferimento a superclasse*** può avvenire anche ***implicitamente*** (come tra **int** e **double**)

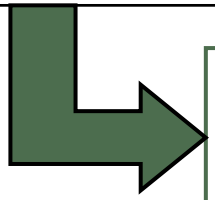
```
BankAccount other = new BankAccount(1000);
SavingsAccount sAcct = new SavingsAccount(10);
other.transferTo(sAcct, 500);
```

- Il compilatore sa che il metodo **transferTo** richiede un riferimento di tipo **BankAccount**, quindi
  - controlla che **sAcct** sia un riferimento ad una classe derivata da **BankAccount**
  - effettua la conversione automaticamente

Conversione fra riferimenti

- La conversione inversa non può invece avvenire automaticamente

```
BankAccount bAcct = new BankAccount(1000);
SavingsAccount sAcct = bAcct;
```



```
incompatible types
found   : BankAccount
required: SavingsAccount
       sAcct = bAcct;
                ^
1 error
```

- ***Ma ha senso cercare di effettuare una conversione di questo tipo?***

Conversione fra riferimenti

- La conversione di un riferimento a ***superclasse*** in un riferimento a ***sottoclasse*** ha senso soltanto se, per le specifiche dell'algoritmo, siamo sicuri che il riferimento a ***superclasse*** punta in realtà ad un oggetto della ***sottoclasse***
  - richiede un ***cast*** esplicito

```
SavingsAccount sAcct = new SavingsAccount(1000);
BankAccount bAcct = sAcct;
...
SavingsAccount sAcct2 = (SavingsAccount) bAcct;
// se in fase di esecuzione bAcct non punta
// effettivamente ad un oggetto SavingsAccount
// l'interprete lancia ClassCastException
```

Upcasting

```
A a; B b; D d;
d = new D();
```

```
b = d;
```

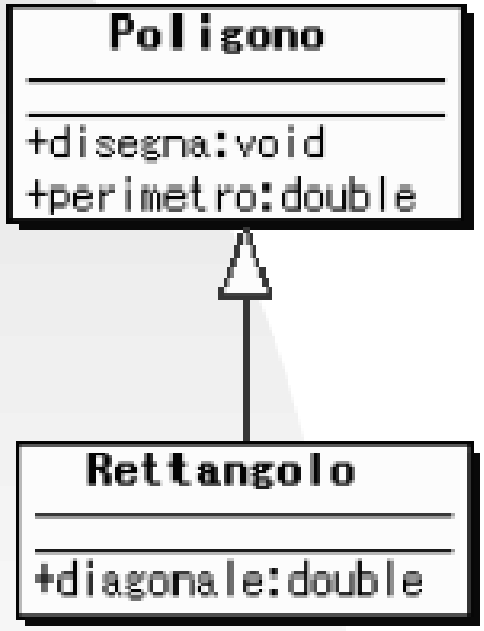
d viene visto come se fosse un oggetto di tipo B

```
a = d;
```

d viene visto come se fosse un oggetto di tipo A

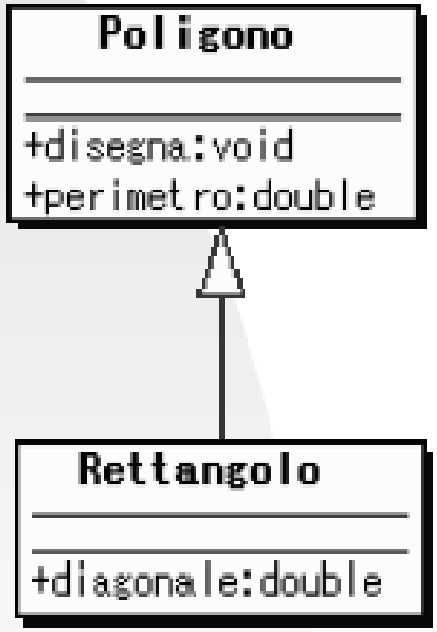
- Upcasting: ci si muove da un tipo specifico ad uno più generico (da un tipo ad un suo "sopratipo").
- L'*upcasting* è sicuro per il type checking: dato che una sottoclasse eredita tutti i metodi delle sue sopraclassi, ogni messaggio che può essere inviato ad una sopraclasse può anche essere inviato alla sottoclasse senza il rischio di errori durante l'esecuzione.

Upcasting




```
Poligono p;
Rettangolo r;
...
p.disegna();
r.disegna();    ⇦  corretto per il
p.perimetro();      compilatore
r.perimetro();  ⇦
```

Upcasting

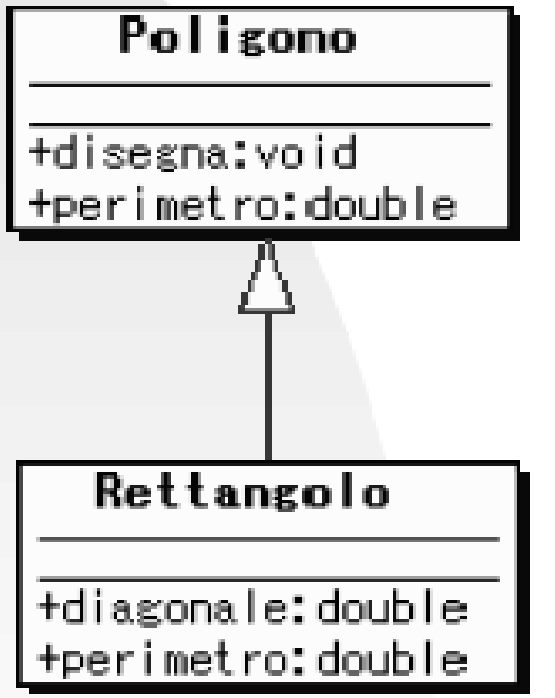



```
Poligono p;
Rettangolo r;
p = r;
r.diagonale(); corretto

p.diagonale();
```

errore di compilazione
p è di tipo *Poligono* e non ha il metodo *diagonale*

se si facesse il controllo a runtime, sarebbe corretto perché p è legata ad un rettangolo (che possiede il metodo *diagonale*).

Upcasting




```
Poligono p;
Rettangolo r;
...
p = r;
p.disegna();

p.perimetro();
```

corretto per il compilatore,
ma
quale metodo si esegue?

Quello di Poligono o
quello di Rettangolo?

Binding dinamico

- Un oggetto decide quale metodo applicare a se stesso in base alla propria posizione nella gerarchia dell'ereditarieta`
- Binding dinamico: decidere a tempo di esecuzione quale metodo applicare
- Binding statico: decidere a tempo di compilazione quale funzione applicare (Pascal, C, …)

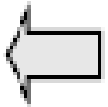
Java adotta il binding dinamico

Binding dinamico

```
p.perimetro();
```

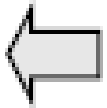
Si esegue il metodo *perimetro* dell'oggetto a cui *p* fa riferimento in quel momento

```
Poligono p = new Poligono();
Rettangolo r = new Rettangolo();

p.perimetro();
```



si esegue il metodo *perimetro* di Poligono

```
p = r;

p.perimetro();
```



si esegue il metodo *perimetro* di Rettangolo

Binding dinamico

- In questo contesto:
- Binding: legame fra il nome di un metodo in una invocazione e (codice del) metodo.
- obj.m(): quale metodo m viene eseguito?
- Nei linguaggi tradizionali le chiamate di procedura vengono risolte dal compilatore.
- Nei linguaggi ad oggetti (tranne il C++) le chiamate di metodi sono risolte dinamicamente.
- BINDING DINAMICO: la forma di un oggetto determina dinamicamente quale versione di un metodo applicare.

Polimorfismo

Polimorfismo

- Sappiamo già che un oggetto di una sottoclasse può essere usato come se fosse un oggetto della superclasse

```
BankAccount acct = new SavingsAccount(10);
acct.deposit(500);
acct.withdraw(200);
```

- Quando si invoca **deposit**, quale metodo viene invocato?
  - il metodo **deposit** definito in **BankAccount**?
  - il metodo **deposit** ***ridefinito*** in **SavingsAccount**?

Polimorfismo

- Questa semantica si chiama ***polimorfismo*** ed è caratteristica dei linguaggi OOP

  l'invocazione di un metodo è sempre determinata dal tipo dell'oggetto effettivamente usato come parametro implicito**, e NON dal tipo della variabile oggetto**

- Si parla di polimorfismo (dal greco, "molte forme") perché ***si compie la stessa elaborazione*** (**deposit**) ***in modi diversi***, dipendentemente dall'oggetto usato

Meccanismi per ottenere il Polimorfismo

- ***Overloading***
  - l'invocazione del metodo **println** si traduce in realtà nell'invocazione di un metodo scelto fra alcuni metodi diversi, in relazione al tipo del parametro esplicito
  - ***il compilatore decide quale metodo invocare***
  - ***selezione anticipata*** (***early binding***)
- Overriding

  In questo caso la situazione è molto diversa, perché ***la decisione non può essere presa dal compilatore, ma deve essere presa dall'ambiente runtime***
  - si parla di ***selezione posticipata*** (***late binding***)

Intervallo

Altri conti bancari

- Vogliamo un nuovo tipo di libretto di risparmio che preveda interessi più alti, ma vincoli il cliente a non prelevare denaro prima di un certo tempo.
  Eventuali prelievi anticipati vengono penalizzati.

- Inoltre vogliamo un conto corrente che garantisca un certo numero di operazioni gratuite. Le operazioni che eccedono tale soglia sono soggette ad un costo.

```
public class TimeDepositAccount extends SavingsAccount
{  public TimeDepositAccount(double rate, int maturity)
   {  super(rate);
      periodsToMaturity = maturity; }
   public void addInterest()
   {  periodsToMaturity--;
      super.addInterest();}
   public void withdraw(double amount)
   {  if (periodsToMaturity > 0)
            super.withdraw(EARLY_WITHDRAWAL_PENALTY);
      super.withdraw(amount); }
   private int periodsToMaturity;
   private static double EARLY_WITHDRAWAL_PENALTY = 20;  }
```

super si usa per accedere a dati e funzioni della superclasse

```
public class CheckingAccount extends BankAccount
{
 public CheckingAccount(int initialBalance)
 { super(initialBalance);
   transactionCount = 0; }
 public void deposit(double amount)
 {  transactionCount++;
   super.deposit(amount); }
 public void withdraw(double amount)
 {  transactionCount++;
   super.withdraw(amount); }
 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;}
```

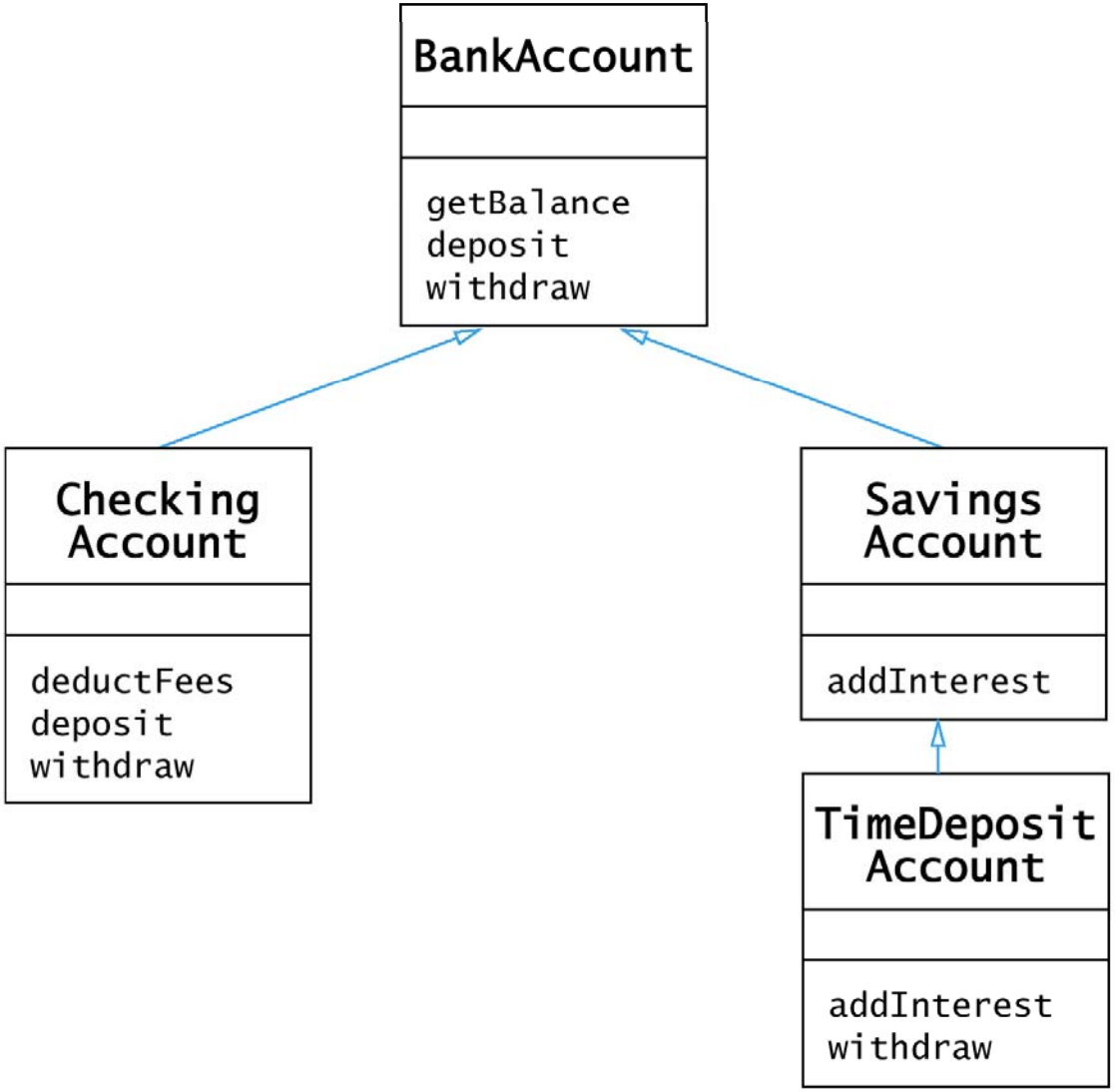
Programma AccountTest.java


```
public class AccountTest
{
 public static void endOfMonth(SavingsAccount savings)
 {  savings.addInterest();
 }

 public static void endOfMonth(CheckingAccount checking)
 {  checking.deductFees();
 }

 public static void printBalance(String name,
                                 BankAccount account)
 {  System.out.println("Il saldo del conto " + name
       + " è di $" + account.getBalance());
 }
```

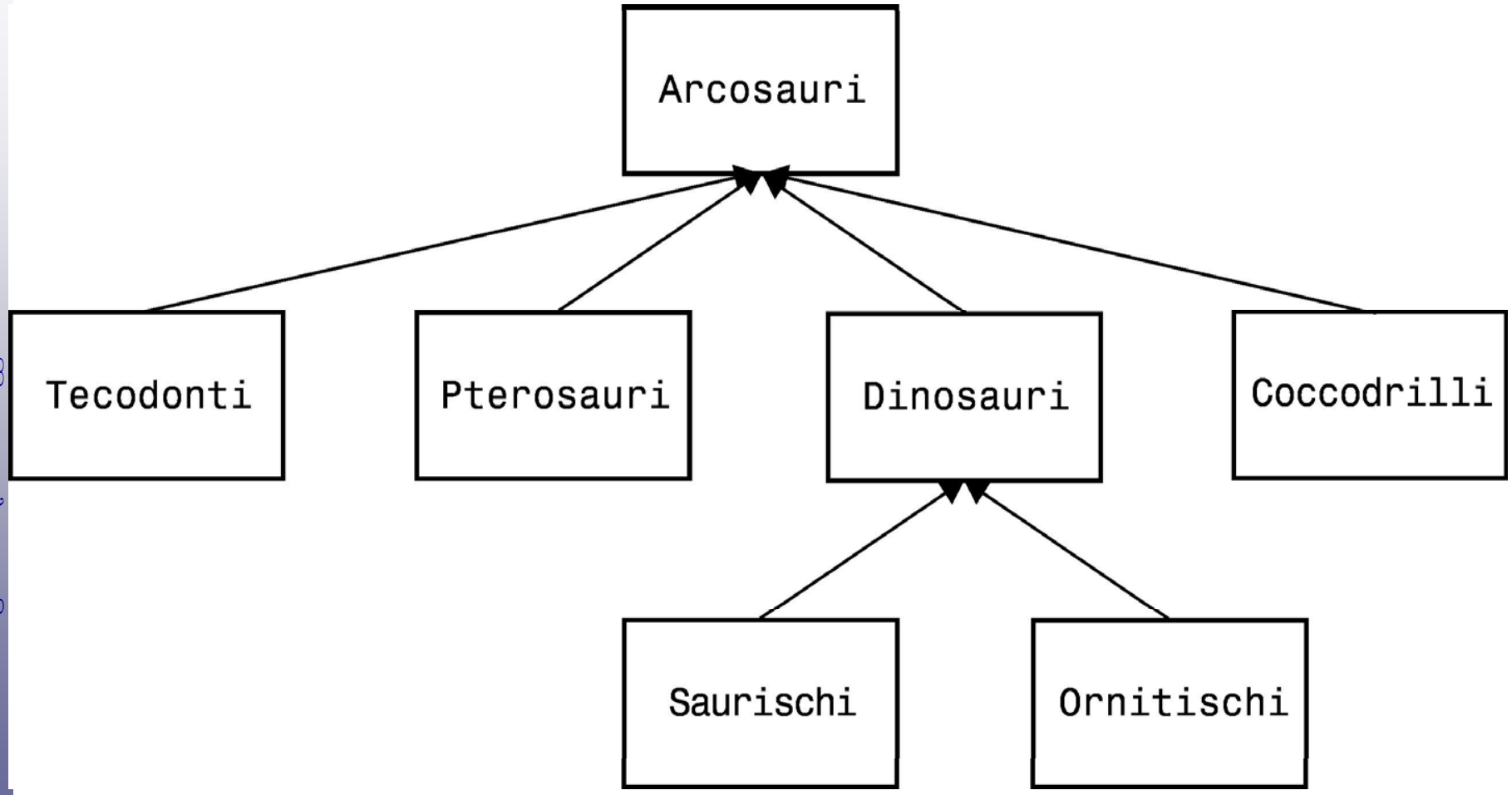
Programma AccountTest.java (continua)

```
public static void main(String[] args)
{ SavingsAccount momsSavings = new SavingsAccount(0.5);
  TimeDepositAccount collegeFund=new TimeDepositAccount(1,3);
  CheckingAccount harrysChecking = new CheckingAccount(0);

  momsSavings.deposit(10000);
  collegeFund.deposit(10000);
  momsSavings.transfer(harrysChecking, 2000);
  collegeFund.transfer(harrysChecking, 980);
  harrysChecking.withdraw(500);
  harrysChecking.withdraw(80);
  endOfMonth(momsSavings);
  endOfMonth(collegeFund);
  endOfMonth(harrysChecking);

  printBalance("Risparmi della mamma", momsSavings);
  printBalance("Fondo per il college", collegeFund);
  printBalance("Conto di Harry", harrysChecking);
 } // Fine della funzione main
} // Fine della classe AccountTest
```

La gerarchia dei conti correnti



La gerarchia dei rettili antichi



Gerarchia di componenti per la grafica