Ereditarietà (ancora)

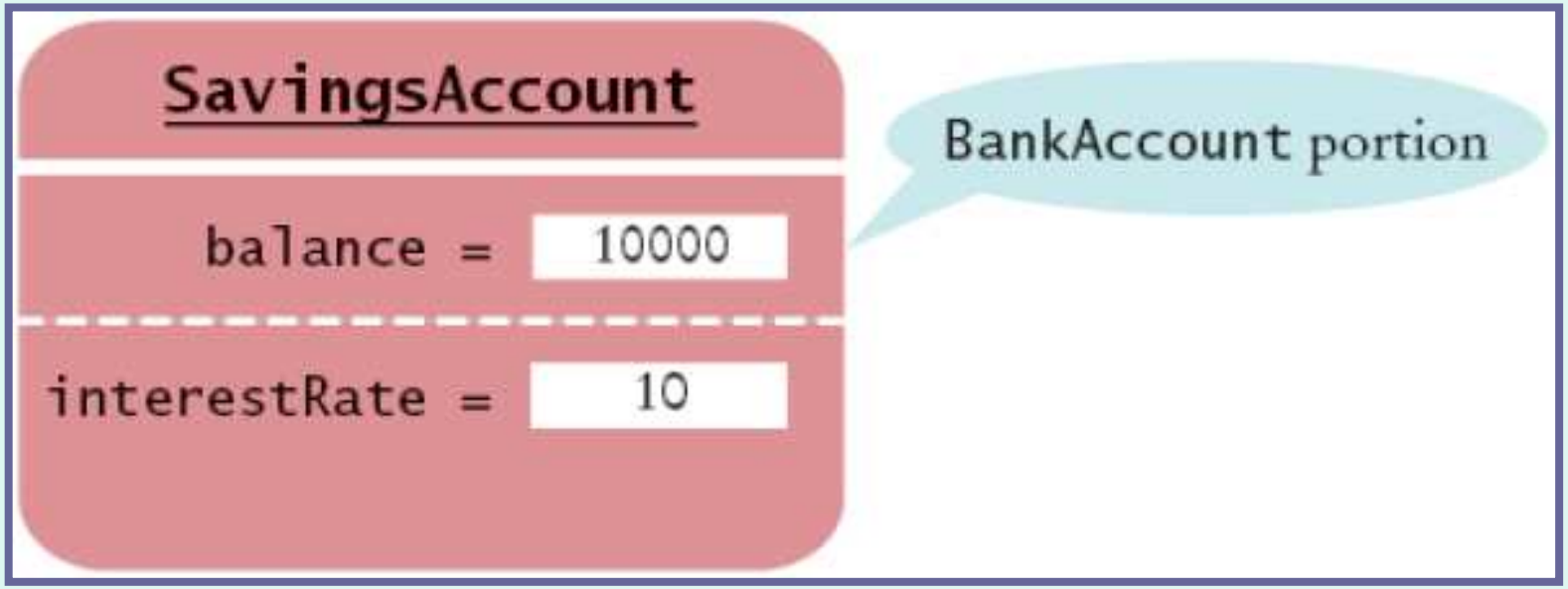
Costruttori di sottoclasse

- **Il costruttore di una sottoclasse provvede ad inizializzare la struttura delle istanze della sottoclasse**
- **Come abbiamo visto, questa include la parte definita nella superclasse**
- **Per inizializzare i campi privati della superclasse invochiamo il costruttore della superclasse**

Continua…

Istanze di sottoclasse

- **Un oggetto di tipo `SavingsAccount` eredita il campo `balance` da `BankAccount`, ed ha un campo aggiuntivo, `interestRate`:**



Costruttori di sottoclasse

- **Per invocare il costruttore della superclasse dal costruttore di una sottoclasse usiamo la parola chiave `super` seguita dagli argomenti**
- **Deve essere il primo comando del costruttore della sottoclasse**

Continua…

Costruttori di sottoclasse

```
class SavingsAccount extends BankAccount
{
   public SavingsAccount(double balance, double ir)
   {
      // Chiamata al costruttore di superclasse
      super(balance);


      // inizializzazioni locali
      interestRate = ir;
   }
   . . .
}
```

- NB: non confondere con la chiamata di metodo della superclasse `super.m(…)`

Continua…

Costruttori di sottoclasse

- **Se il costruttore di sottoclasse non invoca esplicitamente il costruttore di superclasse, il compilatore inserisce la chiamata `super()` al costuttore di default (senza parametri)**
  - Se tutti i costruttori della superclasse richiedono parametri, errore di compilazione

Conversioni di tipo

- Una sottoclasse è un sottotipo della sua superclasse
- Possiamo assegnare ad una variabile di classe un riferimento di tipo sottoclasse

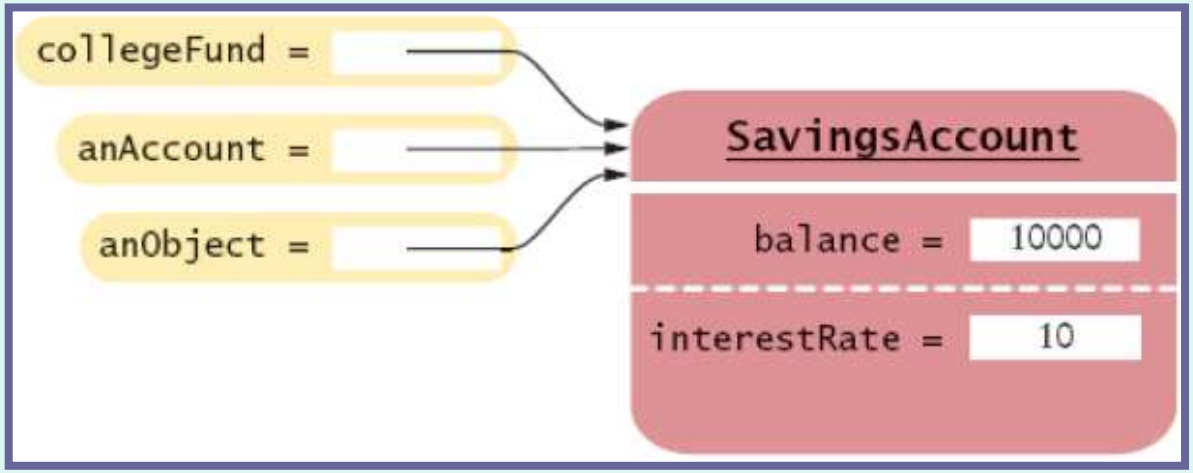
```
SavingsAccount collegeFund = new SavingsAccount(10);

BankAccount anAccount = collegeFund;

Object anObject = collegeFund;
```

Conversioni di tipo

- I tre riferimenti contenuti in `collegeFund`, `anAccount` e `anObject` riferiscono tutti lo stesso oggetto, di tipo `SavingsAccount`



Conversioni di tipo

- **Utilizzare un riferimento di tipo superclasse causa una perdita di informazione**

```
BankAccount anAccount = new SavingsAccount(10);

anAccount.deposit(1000); // OK
// deposit() e' un metodo di BankAccount

anAccount.addInterest(); // Compiler Error
// addInterest() non e' un metodo di BankAccount
```

Cast

- **Possiamo recuperare informazione utilizzando cast**

```
BankAccount anAccount = new SavingsAccount(10);

anAccount.deposit(1000); // OK
// deposit() e' un metodo di BankAccount

((SavingsAccount)anAccount).addInterest(); // OK
```

- **Attenzione:**
  - se **`anotherAccount`** non punta ad un **`SavingsAccount`** ClassCastException a run

Conversioni di tipo

- **Soluzione: utilizzo dell'operatore `instanceof`**

```
if (anAccount instanceof SavingsAccount)
{
   ((SavingsAccount)anAccount).addInterest(); // OK

   . . .
}
```

- **verifica se `anAccount` punta ad un oggetto di tipo `SavingsAccount` (o di un sottotipo)**

Conversioni di tipo

- **Nulla di nuovo …**
  - stesse idee e meccanismi visti quando abbiamo parlato delle interfacce.

Polimorfismo

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

- Questo codice può essere utilizzato per trasferimenti su qualunque **`BankAccount`**

Polimorfismo – *dynamic dispatch*

- **Meccanismo già discusso per le interfacce**
- **Ogni variabile ha due tipi**
  - *Statico*: il tipo dichiarato
  - *Dinamico*: il tipo dell'oggetto a cui la variabile riferisce
- **Il metodo invocato da un messaggio dipende dal tipo dinamico della variabile, non dal tipo statico**

Continua…

Polimorfismo – *dynamic dispatch*

- **Il compilatore verifica che esista un metodo da selezionare in risposta al messaggio**
  - se non esiste errore
  - se esiste decide la firma del metodo da invocare
- **Il corpo del metodo, con il tipo selezionato, viene determinato a run time.**

Polimorfismo – *dynamic dispatch*

- **Dynamic dispatch al lavoro**

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

- **La selezione (dispatch) dei metodi `withdraw()` e `deposit()` da eseguire dipende dal tipo dinamico di `this` e di `other`, rispettivamente**

Continua…

Polimorfismo – *dynamic dispatch*

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

```
BankAccount sa = new SavingsAccount(10);
BankAccount ca = new CheckingAccount();
sa.transfer(1000, ca);
```

- **invoca**
  - **`SavingsAccount.withdraw()` su `sa`**
  - **`CheckingAccount.deposit()` su `ca`**

Continua…

File BankAccount.java

```
01: /**
02:    La classe radice della gerarchia
03:
04: */
05: public class BankAccount
06: {
07:    /**
08:       Constructs a bank account with a zero balance.
09:    */
10:    public BankAccount()
11:    {
12:       balance = 0;
13:    }
14:
15:    /**
16:       Constructs a bank account with a given balance.
17:       @param initialBalance the initial balance
18:    */
```

Continua...

File BankAccount.java

```
19:    public BankAccount(double initialBalance)
20:    {
21:       balance = initialBalance;
22:    }
23:
24:    /**
25:       Peposita un importo sul conto.
26:       @param amount l'importo da depositare
27:    */
28:    public void deposit(double amount)
29:    {
30:       balance = balance + amount;
31:    }
32:
33:    /**
34:       Prelievo di un importo dal conto.
35:       @param amount l'importo da prelevare
36:    */
```

Continua...

File BankAccount.java

```
37:    public void withdraw(double amount)
38:    {
39:       balance = balance - amount;
40:    }
41:
42:    /**
43:       Saldo corrente del conto.
44:       @return il valore del campo balance
45:    */
46:    public double getBalance()
47:    {
48:       return balance;
49:    }
50:
51:    /**
52:       Trasferimento da questo conto ad un altro conto
53:       @param amount l'importo da trasferire
54:       @param other il conto su cui trasferire
55:    */
```

Continua...

File BankAccount.java

```
56:    public void transfer(double amount, BankAccount other)
57:    {
58:       withdraw(amount);
59:       other.deposit(amount);
60:    }
61:
62:    private double balance;
63: }
```

File CheckingAccount.java

```
01: /**
02:    Un conto corrente, con commissioni sulle operazioni
03: */
04: public class CheckingAccount extends BankAccount
05: {
06:    /**
07:       Costruisce un conto con un saldo iniziale.
08:       @param initialBalance il saldo iniziale
09:    */
10:    public CheckingAccount(double initialBalance)
11:    {
12:       // costruttore della superclasse
13:       super(initialBalance);
14:
15:       // inizializza i campi locali
16:       transactionCount = 0;
17:    }
18:
```

Continua...

File CheckingAccount.java

```
19:    public void deposit(double amount)
20:    {
21:       transactionCount++;
22:       // deposita invocando il metodo della superclasse
23:       super.deposit(amount);
24:    }
25:
26:    public void withdraw(double amount)
27:    {
28:       transactionCount++;
29:       // preleva invocando il metodo della superclasse
30:       super.withdraw(amount);
31:    }
32:
33:    /**
34:       Deduce le commissioni accumulate e riazzera il
35:       contatore delle transazioni.
36:    */
```

Continua...

File CheckingAccount.java

```
37:    public void deductFees()
38:    {
39:       if (transactionCount > FREE_TRANSACTIONS)
40:       {
41:          double fees = TRANSACTION_FEE *
42:                (transactionCount - FREE_TRANSACTIONS);
43:          super.withdraw(fees);
44:       }
45:       transactionCount = 0;
46:    }
47:
48:    private int transactionCount;
49:
50:    private static final int FREE_TRANSACTIONS = 3;
51:    private static final double TRANSACTION_FEE = 2.0;
52: }
```

File SavingsAccount.java

```
01: /**
02:    Un libretto bancario con interessi fissi.
03: */
04: public class SavingsAccount extends BankAccount
05: {
06:    /**
07:       Costruisce un libretto con un tasso di interesse.
08:       @param rate il tasso di interesse
09:    */
10:    public SavingsAccount(double rate)
11:    {
12:       interestRate = rate;
13:    }
14:
15:    /**
16:       Aggiunge gli interessi maturati al conto=.
17:    */
```

Continua...

File SavingsAccount.java

```
18:    public void addInterest()
19:    {
20:       double interest = getBalance() * interestRate / 100;
21:       deposit(interest);
22:    }
23:
24:    private double interestRate;
25: }
```

File AccountTester.java

```
01: /**
02:    Test per la classe BankAccount e le sue sottoclassi
03:
04: */
05: public class AccountTester
06: {
07:    public static void main(String[] args)
08:    {
09:       SavingsAccount momsSavings
10:             = new SavingsAccount(0.5);
11:
12:       CheckingAccount harrysChecking
13:             = new CheckingAccount(100);
14:
15:       momsSavings.deposit(10000);
16:
```

Continua...

File AccountTester.java

```
17:         momsSavings.transfer(2000, harrysChecking);
18:         harrysChecking.withdraw(1500);
19:         harrysChecking.withdraw(80);
20:
21:         momsSavings.transfer(1000, harrysChecking);
22:         harrysChecking.withdraw(400);
23:
24:         // test per le operazioni di fine mese
25:         momsSavings.addInterest();
26:         harrysChecking.deductFees();
27:
28:         System.out.println("Mom's savings balance = $“
29:               + momsSavings.getBalance());
30:
31:         System.out.println("Harry's checking balance = $“
32:               + harrysChecking.getBalance());
33:     }
34: }
```

Invocazione di metodi

- **Due fasi nella selezione del metodo da invocare:**
- **Fase statica:**
  - seleziona il tipo del metodo, in funzione del tipo statico degli argomenti presenti nel messaggio
  - risoluzione dell'overloading
- **Fase dinamica: dispatch**
  - seleziona il corpo del metodo, in funzione del tipo dinamico del destinatario dell'invocazione

Selezione statica

`exp.m(a1,..., an)`

Due fasi:

1. Determina il tipo di **`exp`**
2. Seleziona la firma **`T m(T1,…Tn)`** del metodo da invocare in base al tipo degli argomenti **`a1,...,an`**

In questa fase (statica)

- tipo = tipo statico

Selezione Statica / destinatario

`exp.m(a1,..., an)`

- Determina il tipo statico **`S`** di **`exp`**:

  1. **`exp = super:`**
     - **`S`** è la superclasse della classe in cui l'invocazione occorre:
  2. **`exp = this`**:
     - **`S`** è la classe in cui l'invocazione occorre
  3. in tutti gli altri casi:
     - **`S`** è il tipo dichiarato per **`exp`**

Selezione statica / metodo

`exp.m(a1,..., an)` **`exp:S`**

- Determina la firma del metodo da invocare
  1. calcola il tipo degli argomenti, **`a1:S1,.... an:Sn`**
  2. determina il *best match* **`m(T1,...,Tn)`** per l'invocazione **`m(a1:S1,… an:Sn),`** a partire da **`S`**
- Risoluzione overloading
  - se trovi una sola firma: ok
  - altrimenti se nessuno / più di *best match:* errore

Esempio

```
class A {
   public void m(double d){System.out.println("A.m(double)");
   public void m(int i) {  System.out.println(“A.m(int)"); }
 }
class B extends A {
   public void m(double d){System.out.print(“B.m(double)”); }
}
class over {
  public static void main(String[] args) {
  { A a = new B();  a.m(1.5); }
}
 // Selezione statica: BEST(A, m(double)) = { A.m(double) }
 // Dispatch: B ridefinisce m(double). Quindi esegui  B.m(double)
```

Esempio

```
class A {
   public void m(double d){ System.out.println("A.m(double)");
   public void m(int i) {  System.out.println(“A.m(int)"); }
 }
class B extends A {
  public void m(double d){System.out.print(“B.m(double)”); }
}
class over {
  public static void main(String[] args) {
  { A a = new B();  a.m(1); }
}
   // Selezione statica: BEST(A, m(int)) = { A.m(int) }
   // Dispatch: B non ridefinisce m(int). Quindi esegui  A.m(int)
```

Esempio

```
class A {
   public void m(double d){ System.out.println("A.m(double)");
   public void m(int i) {  System.out.println(“A.m(int)"); }
 }
class B extends A {
  public void m(double d){System.out.print(“B.m(double)”); }
}
class over {
  public static void main(String[] args) {
  { B b = new B();   b.m(1); }
}
   // Selezione statica: BEST(B, m(int)) = { A.m(int) }
   // Dispatch: B non ridefinisce m(int). Quindi esegui  A.m(int)
```

Esempio

```
class A {
    public void m(int i)
    {  System.out.println("A.m(int)"); }
}
class B extends A {
     public void m(String s)
     {  System.out.println("B.m(String)"); }
}
class over {
    public static void main(String[] args) {
        B b = new B();
        A a = new B();
        a.m(1)              // BEST(A,m(int))   = {A.m(int)}
        b.m(“a string”)     // BEST(B,m(String))= {B.m(String)}
        a = b;
        a.m(“a string”);    // BEST(A,m(string))= {}
    }
}
```

Esempio

```
class A {
        public void m(int i, float f) { /* just return */}
        public void m(float f, int i) { /* just return */}
}

class test {
        public static void main(String[] args) {
        A a = new A();
        a.m(1, 1);
}
// BEST(A,m(int,int)) = { A.m(int,float), A.m(float,int) }
```