Il Linguaggio Java

Ereditarietà

Ereditarietà

- L'ereditarietà permette di creare nuove classi sulla base di classi esistenti
- In particolare, permette di
  - **riusare** il codice (metodi e campi);
  - **aggiungere** nuovi metodi e nuovi campi;
  - **ridefinire** metodi e campi esistenti (overriding)

Parola chiave **extends**

```
class A {
    // ...
}
class B extends A {
    // Differenze rispetto ad A
}
```

Classe A: *classe base*, *classe padre* o *superclasse*

Classe B:*classe derivata*, *classe figlio,* o *sottoclasse*

Ereditarietà

- La sottoclasse
  - eredita tutti i membri della superclasse
  - può aggiungere nuovi membri
  - può ridefinire i metodi della superclasse
  - non può accedere ai membri privati della superclasse

Esempio: conti bancari

Un **SavingsAccount** (conto di risparmio)

- **è un** **BankAccount** (conto bancario), che
- frutta un tasso di interesse fisso sui depositi

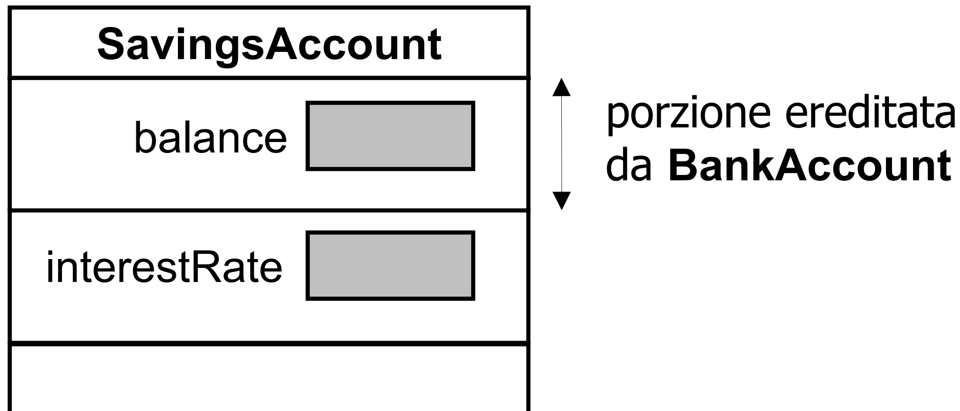
Rispetto a **BankAccount**, **SavingsAccount**

- ha un campo **interestRate** che specifica il tasso di interesse
- ha un metodo **addInterestRate** che applica il tasso di interesse
- ha un costruttore che imposta il valore iniziale del tasso di interesse

La classe **SavingsAccount**

```
class SavingsAccount extends BankAccount {
  public SavingsAccount(double aRate) {
    // ...
  }
  public void addInterest() {
    // ...
  }
   private double interestRate;
}
```

Oggetto di una sottoclasse



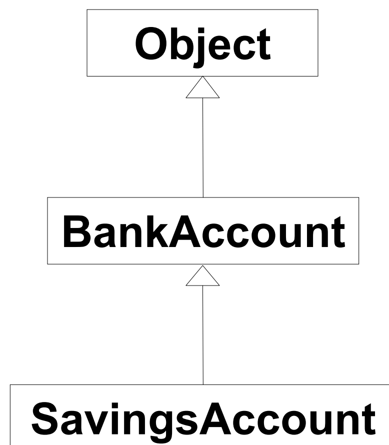
Accesso ai metodi della superclasse

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

Le chiamate ai metodi della superclasse (**getBalance**, **deposit**) utilizzano il parametro implicito **this** del metodo **addInterest**

La classe Object

Una classe **A** che non estende esplicitamente un'altra classe costituisce una sottoclasse della classe **Object**



Principali metodi di **Object**

- **toString**
- **clone**
- **equals**

Conversioni tra tipi riferimento (I)

ESTENSIONE DI TIPO RIFERIMENTO

S → **T**

purché S sia una sottoclasse di T (**S** *è un* **T**)

class T {/* ... */}

class S extends T {/*…}

La conversione da **T** verso **S** è una *riduzione*

Conversioni tra tipi di classi (II)

- **Assegnamento ed invocazione di metodo**: sono consentite solo le conversioni identità e per estensione
- Esempio

T t = new T();

S s = new S();

t = s; // S *è un* T

Conversioni tra tipi di classi (II)

- **Casting** consente la conversione identità e le conversioni per estensione e per riduzione
- Esempio

T t = new T();

S s;

s = (S)t; // Si forza **t** ad essere un **S**. Pericoloso!!

Conversioni tra tipi di classi (III)

```
SavingsAccount mio = new SavingsAccount(10);
BankAccount unConto = mio; // OK
Object unOggetto = mio; // OK

mio.addInterest(); // OK
unConto.deposit(20);    // OK
unConto.addInterest();  // compile-time error
unOggetto.deposit(); // compile-time error
```

- le tre variabili oggetto **mio**, **unConto** ed **unOggetto** riferiscono lo stesso oggetto…
- ma **unConto** non “sa tutto” dell’oggetto **mio** ed
- **unOggetto** ne sa ancora meno…
- ma allora a cosa serve???

Riuso del codice della superclasse

```
SavingsAccount mio = new SavingsAccount(10);
BankAccount tuo = new BankAccount(10);
…
tuo.transfer(mio, 5.0);
```

al metodo **transfer** non serve sapere tutto dell’oggetto **mio**, gli basta sapere che è un **BankAccount**

Cast tra tipi riferimento

```
SavingsAccount mio = new SavingsAccount(10);
…
Object unOggetto = mio; // mio è un Object
…
SavingsAccount unConto = (savingsAccount)unOggetto;
…
unConto.addInterest(); // OK
…
```

Se, però, **unOggetto** non riferisce un oggetto istanza di **SavingsAccount**, l'invocazione del metodo produce un errore a run-time

Parola chiave **instanceof**

Conviene sempre verificare la fattibilità di un cast di tipo riferimento

```
if ( unOggetto instanceof SavingsAccount )
        unConto = (SavingsAccount)unOggetto;
else
        unConto = null;
```

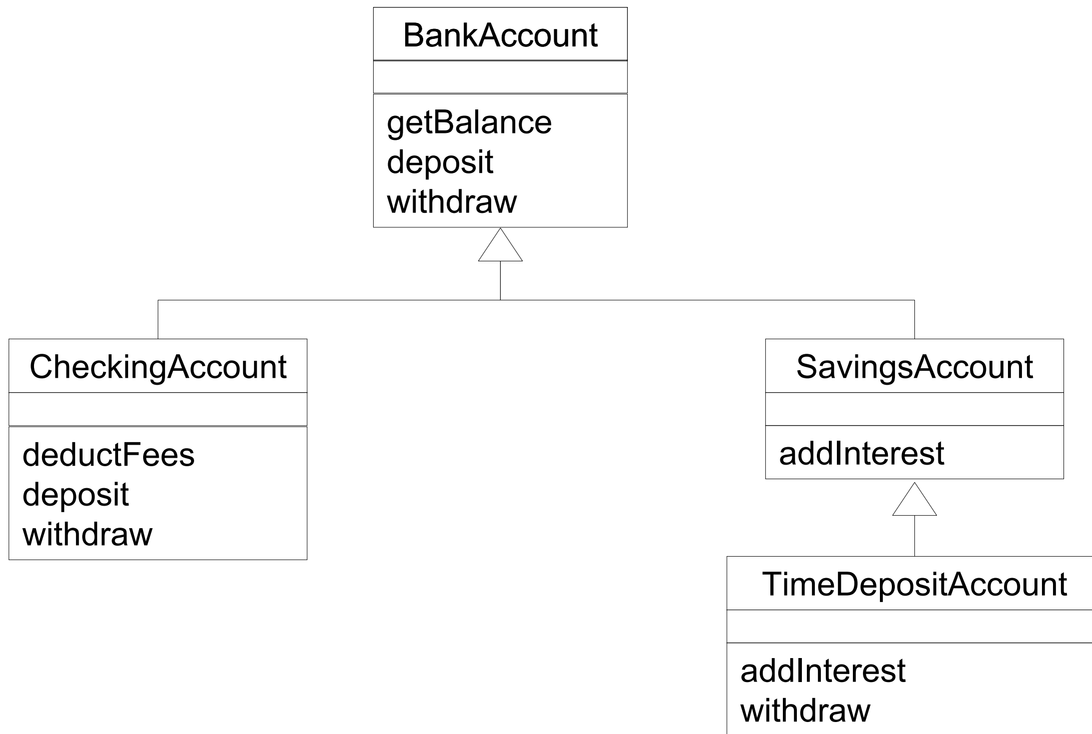
Gerarchie di ereditarietà (I)

- Il conto corrente (**CheckingAccount**) è un conto bancario (**BankAccount**) che
  - non ha interessi,
  - offre un numero limitato di operazioni mensili gratuite ed
  - addebita una commissione per ciascun movimento aggiuntivo
- Il libretto di risparmio (**SavingsAccount**) è un conto bancario che
  - frutta interessi mensili

Gerarchie di ereditarietà (II)

- Il conto vincolato (**TimeDepositAccount**) è un conto di risparmio (**SavingsAccount**) che
  - impegna a lasciare il denaro nel conto per certo numero di mesi e
  - prevede una penale per un prelievo anticipato

Gerarchie di ereditarietà (III)



Metodi nella sottoclasse

- La sottoclasse può
  - **ereditare** uno o più membri della superclasse
  - **definire** nuovi metodi (membri)
  - **sovrascrivere** (ridefinire) uno o più metodi della superclasse

Metodi nella sottoclasse (I)

- **Ereditare** un metodo dalla superclasse
  - Se un metodo della superclasse non viene sovrascritto (ridefinito) allora tale metodo viene **ereditato** e può essere applicato agli oggetti della sottoclasse
- Esempio
  - La classe **CheckingAccount** eredita il metodo **BankAccount.getBalance** dalla superclasse **BankAccount**

Metodi nella sottoclasse (II)

- **Definire** nuovi metodi nella sottoclasse
  - Se nella sottoclasse si **definisce** un metodo che non esiste nella superclasse allora tale metodo può essere applicato solo agli oggetti della sottoclasse
- Esempio
  - Il metodo **SavingsAccount.addInterest**

Metodi nella sottoclasse (III)

- **Sovrascrivere** (**override**) un metodo della superclasse
  - Nella sottoclasse si definisce un metodo **m2** con la ***stessa firma*** del metodo **m1** della superclasse
  - Il metodo **m2 sovrascrive m1**, cioè quando si applica il metodo ad un oggetto della sottoclasse viene eseguito **m2**
- Esempio
  - **CheckingAccount.deposit** sovrascrive **BankAccount.deposit**

La classe **CheckingAccount**

```
public class CheckingAccount extends BankAccount {
    public CheckingAccount(int initialBalance) {/* costruttore */}
    public void deposit(double amount) {/* override */}
    public void withdraw(double amount) {/* override */}
    public void deductFees() {/* define */}

    private int transactionCount;
    private static final int FREE_TRANSACTIONS = 3;
    private static final double TRANSACTION_FEE = 3.0;
}
```

La classe CheckingAccount

- **public void deposit(double amount)** accredita **amount** su questo conto corrente ed incrementa il numero di transazioni eseguite
- **public void withdraw(double amount)** preleva **amount** da questo conto corrente ed incrementa il numero di transazioni eseguite
- **public void deductFees()** addebita la commissione complessiva ottenuta applicando la commissione **TRANSACTION_FEE** ad ogni transazione in esubero rispetto a **FREE_TRANSACTIONS** e reimposta a zero il numero di transazioni eseguite

La parola chiave **super**

```
public class CheckingAccount extends
                                    BankAccount {
  public void deposit(double amount) {
    transactionCount++;
    deposit(amount);     // Ricorsione infinita!!
}
```

La parola chiave **super**

```
public class CheckingAccount extends
                                        BankAccount {
   public void deposit(double amount) {
      transactionCount++;
      super.deposit(amount);  // OK!!
}
```


La parola chiave **super**

```
public void deductFees()   {
   if (transactionCount > FREE_TRANSACTIONS)
{
      double fees = TRANSACTION_FEE *
            (transactionCount –
                  FREE_TRANSACTIONS);
      super.withdraw(fees);
   }
   transactionCount = 0;
}
```

Differenza tra **super** e **this**

- La parola chiave **this**
  - costituisce un riferimento al parametro implicito;
  - come prima istruzione, permette di invocare un altro costruttore della stessa classe
- La parola chiave **super**
  - non costituisce un riferimento ad un oggetto
  - permette di invocare un metodo sovrascritto della superclasse
  - come prima istruzione, permette di invocare un costruttore della superclasse

Costruzione della sottoclasse

- **Problema**

  nel costruttore della sottoclasse si vuole impostare il valore iniziale delle variabili ereditate, ma

  la sottoclasse non ha accesso alle variabili private della superclasse

- **Soluzione**:

  si invoca il costruttore della superclasse utilizzando la parola chiave **super**

Costruzione della sottoclasse

```
public CheckingAccount(int initialBalance) {
    // si costruisce la superclasse
    super(initialBalance);

    // si inizializza il contatore delle transazioni
    transactionCount = 0;
}
```


Variabili istanza nella sottoclasse (I)

- Ereditare una variabile istanza della superclasse
  - Tutte le variabili istanza della superclasse sono ereditate automaticamente
  - Esempio: **balance** di **BankAccount**
- Definire nuove variabili istanza nella sottoclasse
  - Qualunque nuova variabile definita nella sottoclasse esiste solo per gli oggetti istanza della sottoclasse
  - Esempio: **transactionCount** di **CheckingAccount**

Variabili istanza nella sottoclasse

- Mettere in ombra una variabile della superclasse
  - Se nella sottoclasse **F** si dichiara una variabile istanza **v** con lo stesso nome $\mathcal{N}$ di una variabile istanza **w** della superclasse **P**, allora **v** mette in ombra **w**, cioè
  - ogni volta che un metodo della sottoclasse utilizza il nome $\mathcal{N}$ si riferirà alla variabile **v**

Variabili in ombra

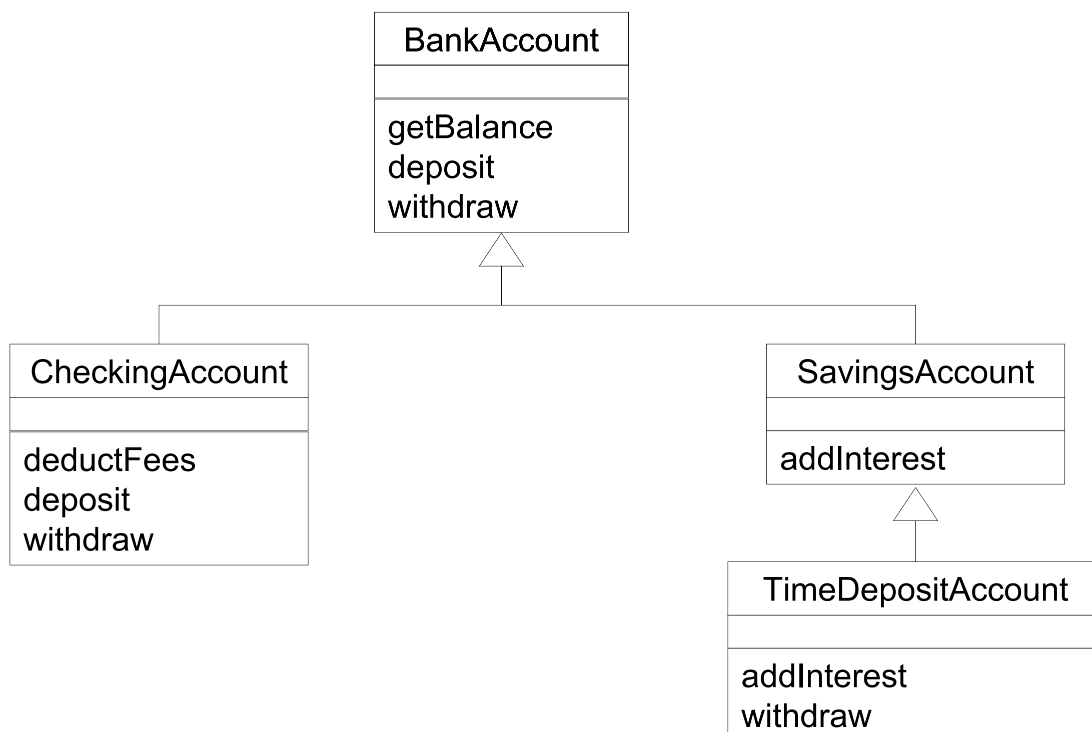
```
public class CheckingAccount extends BankAccount {
    public void deposit(double amount) {
        transactionCount++;
        balance += amount;
 }
    // ...
    private double balance;
}
      …
      CheckingAccount harrys = new CheckingAccount(100);
      …
      harrys.deposit(1500);
      System.out.println(harrys.getBalance()); // 100
```

Costruzione della superclasse

- **Chiamata esplicita**: La chiamata al costruttore della superclasse mediante **super** deve la prima istruzione del costruttore della sottoclasse
- **Chiamata implicita**: Se un costruttore della sottoclasse non chiama esplicitamente un costruttore della superclasse, viene chiamato automaticamente il costruttore di default (senza argomenti) della superclasse

  se la superclasse non ha costruttore di default, il compilatore dà errore

Gerarchie di ereditarietà



Ereditarietà indiretta

- La classe **SavingsAccount** è la superclasse diretta della classe **TimeDepositAccount**
- La classe **BankAccount** è una superclasse indiretta di **TimeDepositAccount**
- **TimeDepositAccount** eredita i metodi **getBalance** e **deposit** da **BankAccount**

  Si possono ereditare metodi da una superclasse indiretta a patto che le superclassi intermedie non li ridefiniscano

Caricamento ed inizializzazione

- In un linguaggio tradizionale,
  - il formato eseguibile di un programma viene caricato interamente;
  - viene eseguita l'inizializzazione della parte statica;
  - infine, il controllo passa al programma

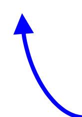
Caricamento ed inizializzazione

- Nel linguaggio Java
  - la classe (il file **.class**) viene caricata al momento del suo primo utilizzo (quando è necessario) e cioè
    - al primo accesso di un membro statico;
    - oppure quando viene creato il primo oggetto della classe
  - nel punto di primo utilizzo di una classe avviene l'inizializzazione dei suoi membri **static**

Polimorfismo (I)

- Un oggetto della sottoclasse è anche un oggetto della superclasse, perciò
- È sempre possibile usare un oggetto della sottoclasse al posto di uno della superclasse

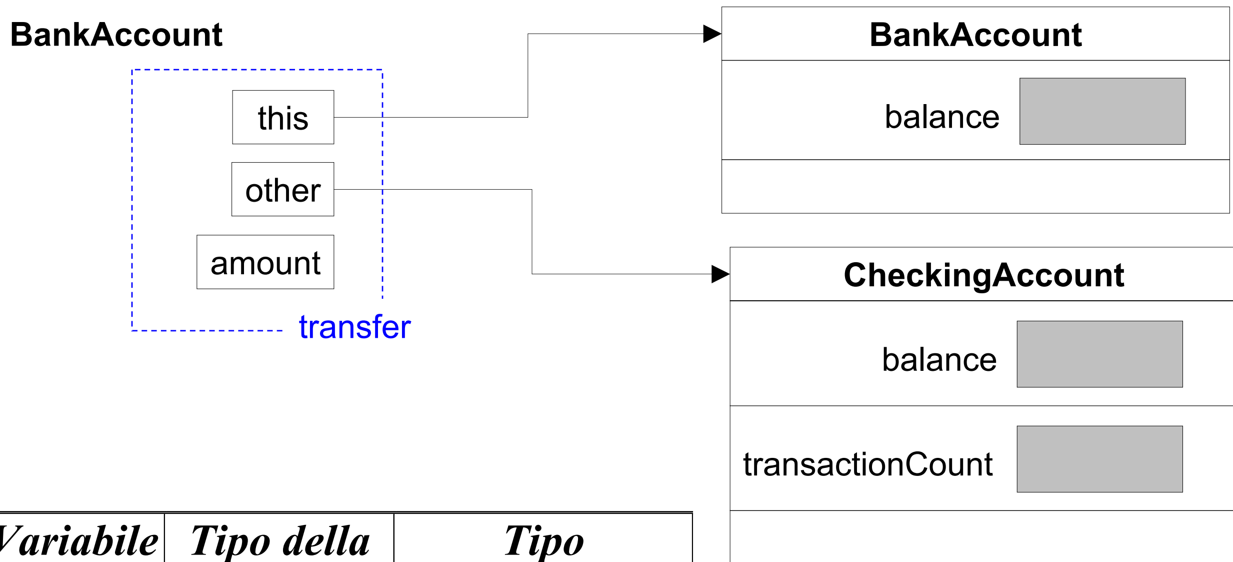
```
BankAccount mio = new BankAccount(100);
CheckingAccount tuo = new CheckingAccount (10);
mio.transfer(tuo, 5);
```



Quale metodo **deposit** viene chiamato?

Si vorrebbe **CheckingAccount.deposit**...

Polimorfismo (II)



Variabile	*Tipo della variabile*	*Tipo dell'oggetto*
this	BankAccount	BankAccount
other	BankAccount	CheckingAccount

Polimorfismo (III)

- Le chiamate di metodo sono sempre determinate dal tipo dell'oggetto effettivo e non dal tipo della variabile riferimento

Perció,

- una stessa chiamata (es., tuo.deposit()) può chiamare metodi diversi;
- il metodo che viene effettivamente chiamato è determinato a tempo di esecuzione (late binding o dynamic binding)

Chiamata di un metodo

Il compilatore risolve **x.f(** ***args*** **)**, con **x** di classe **X**, eseguendo i seguenti passi:

- determina i metodi con la stessa firma **f** ed appartenenti alla classe **X** ed alle sue superclassi;
- seleziona quello più specifico tra quelli accessibili (Se tale metodo non esiste, il compilatore genera un errore)
- se il metodo selezionato è **private**, **static** o **final**, oppure se è un **costruttore**, produce codice per effettuare la chiamata (static/early binding);
- altrimenti, produce codice per: (1) determinare il metodo da invocare e (2) effettuare la chiamata (dynamic/late binding*)*

Dynamic or late binding

- JVM seleziona il metodo da chiamare in base al tipo effettivo dell'oggetto.
  - se l'oggetto è di classe **Y**, derivata da **X**, ed **Y** ha il metodo **f**, allora JVM invoca tale metodo;
  - altrimenti, JVM invoca **X.f**
- È inefficiente eseguire questa selezione ad ogni chiamata di metodo. Perciò, per ogni classe viene costruita la Tabella dei metodi

La Tabella dei metodi