Lezione

- Ereditarietà

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

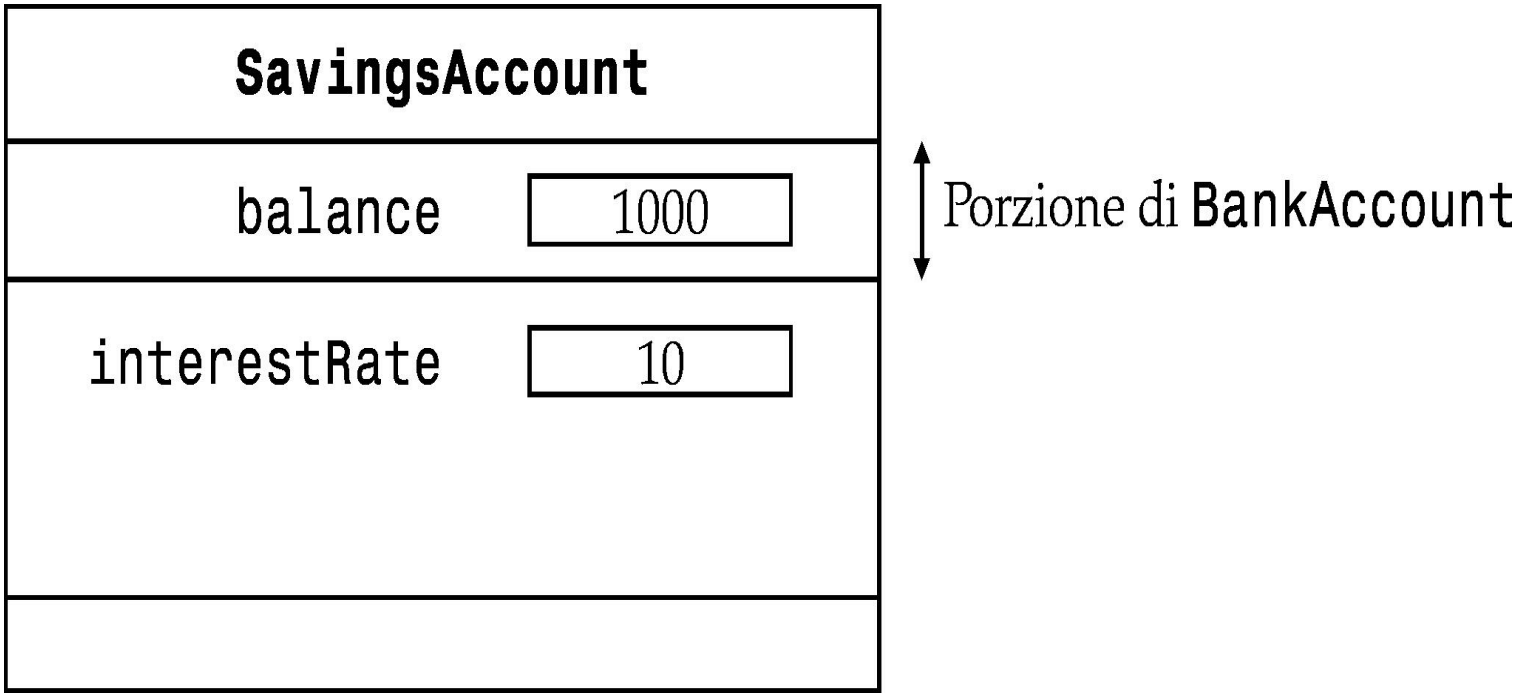
- Supponiamo di voler implementare altri tipi di conto bancario.
- Per esempio, consideriamo un conto bancario che dia interessi ad un certo tasso fisso *interestRate*.
- Dobbiamo riscrivere tutto il codice?
- Possiamo sfruttare il codice esistente estendendolo opportunamente?

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);
   }

}
```

Oggetti della sottoclasse SavingsAccount


```
…
SavingsAccount s = new SavingsAccount(10);
s.deposit(1000);
…
```



Quando si estende una classe...

- Si **ereditano** tutti i metodi e gli elementi di dati della superclasse
- Si possono definire **nuovi metodi**
- Si possono definire **nuovi elementi di dati**
- Si possono **ridefinire metodi** per specializzarne le funzionalità

Ereditare dalla superclasse

- La sottoclasse eredita dalla superclasse e da tutti gli antenati. In particolare eredita:
  - Tutti i membri (variabili e metodi) della superclasse **accessibili** a quella sottoclasse, a meno che
  - Esplicitamente nasconda le variabili o ne sovrascriva i metodi, ridefinendoli.
- I costruttori non sono membri di una classe e perciò non vengono ereditati dalle sottoclassi.

I membri accessibili

- Una sottoclase eredita i membri **accessibili** della superclasse, dichiarati:
  - `public` o `protected` nella superclasse
  - di default, senza modificatore, purchè siano nello stesso pacchetto della superclasse
- Una sottoclasse **non** eredita i membri dichiarati
  - `private` dalla superclasse
  - con lo stesso nome nella sottoclasse
    - Variabili oscurate
    - Metodi sovrascritti

Private, public, protected, o … (Java)

Modificatori d'accesso di Java che controllano la visibilita` di campi di istanze o metodi:

- nessuna specifica: visibile al package
- private: visibile alla sola classe di appartenenza
- public: visibile ovunque
- protected: visibile al package e a tutte le sottoclassi (una classe potrebbe estenderne un'altra non appartenente allo spesso pacchetto)

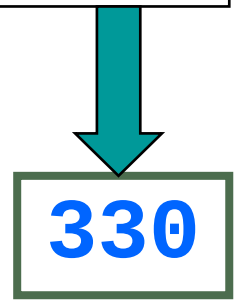
Private, ... (Java)

- Per i metodi:
  - *public* per l'interfaccia
  - *private* per i metodi di "servizio"
- Per i campi:
  - *private* e` sempre la soluzione consigliata, se e` necessario accederne al valore allora si definisce un metodo di accesso
  - *protected* se si vuole renderli accessibili alle sottoclassi
  - *public*, nulla ...

Come usare la classe derivata

- Oggetti della classe derivata **SavingsAccount** si usano come se fossero oggetti di **BankAccount**, con ***qualche proprietà in più***

```
SavingsAccount acct = new SavingsAccount(10);
acct.deposit(500);
acct.withdraw(200);
acct.addInterest();
System.out.println(acct.getBalance());
```



- La classe derivata si chiama
  - ***sottoclasse***
- La classe da cui si deriva si chiama
  - ***superclasse***

Sovrascrivere metodi

- Una classe figlio può *sovrascrivere* la definizione di un metodo ereditato per specializzarlo
  - Un figlio può dover modificare un metodo ereditato
- Il nuovo metodo deve avere **lo stesso prototipo** del metodo della superclasse, ma un diverso corpo
- Il tipo dell'oggetto a cui è inviato il metodo determina la versione del metodo invocato

Sovraccaricare vs. sovrascrivere

- I concetti di ***sovraccaricamento*** e di ***sovrascrittura*** sono diversi

- ***Sovraccaricare*** si riferisce alla possibilità di definire più metodi nella stessa classe con lo stesso nome ma **segnature diverse**
  - Il sovraccaricamento consente di definire operazioni simili in modi diversi con dati diversi
- ***Sovrascrivere*** si riferisce a due metodi, uno della superclasse e uno della sottoclasse, che devono avere lo **stesso prototipo**
  - La sovrascrittura consente di definire operazioni simili in modi diversi per oggetti diversi

Sovrascrivere metodi

- Se un metodo è dichiarato anche con il modificatore **`final`** allora non può essere sovrascritto
- La sovrascrittura può essere applicata a variabili
  - si dice che le variabili sono ***oscurate***
- Il metodo della superclasse può essere esplicitamente invocato mediante il riferimento **`super`**

Costruttori della sottoclasse

- I costruttori non vengono ereditati, non essendo membri di classe
- Vengono definiti esplicitamente dal programmatore
- Oppure viene usato quello di default
  - Il costruttore di default è quello senza argomenti fornito automaticamente quando non ne avete definito uno

Sovraccarico del costruttore

- un secondo costruttore in **SavingsAccount**, per ***aprire un conto corrente di risparmio con saldo iniziale diverso da zero***
  - analogamente a **BankAccount**, che ha due costruttori

```
public class SavingsAccount extends BankAccount
{  public SavingsAccount(double rate)
   {  interestRate = rate;
   }
   public SavingsAccount(double rate,
                         double initialAmount)
   {  interestRate = rate;
      balance = initialAmount; // NON FUNZIONA
   }
   ...
}
```

- Sappiamo già che questo approccio non può funzionare, perché **balance** è una variabile **private** di **BankAccount**

Costruzione della sottoclasse

```
public SavingsAccount(double rate,
                      double initialAmount)
{  interestRate = rate;
   deposit(initialAmount); // POCO ELEGANTE
}
```

- Si potrebbe risolvere il problema simulando un primo versamento, invocando **deposit**
- Questo è lecito, ma non è una soluzione molto buona, perché introduce potenziali effetti collaterali
  - ad esempio, le operazioni di versamento potrebbero avere un costo, dedotto automaticamente dal saldo, mentre il versamento di “apertura conto” potrebbe essere gratuito

Costruzione della sottoclasse

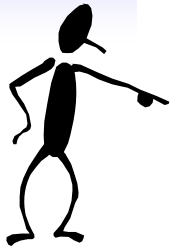
- Dato che la variabile **balance** è gestita dalla classe **BankAccount**, bisogna ***delegare*** a tale classe il compito di inizializzarla
- La classe **BankAccount** ha già un costruttore creato appositamente per gestire l'apertura di un conto con saldo diverso da zero
  - invochiamo tale costruttore dall'interno del costruttore di **SavingsAccount**, usando la sintassi **super(...)**

```
public SavingsAccount(double rate,
                      double initialAmount)
{  super(initialAmount);
   interestRate = rate;
}
```

Costruzione della sottoclasse

- In realtà, un'invocazione di **super** viene sempre eseguita quando la JVM costruisce una sottoclasse
  - se la chiamata a **super(…)** non è indicata esplicitamente dal programmatore, il compilatore inserisce automaticamente l'invocazione di **super( ) *senza parametri***
    - questo avviene, ad esempio, nel caso del primo costruttore di **SavingsAccount**
- L'invocazione ***esplicita*** di **super(…)**, se presente, ***deve essere il primo enunciato del costruttore***

Invocare un costruttore di superclasse



- Sintassi:

```
NomeSottoclasse(parametri)
{   super(eventualiParametri);
    ...
}
```

- Scopo: invocare il costruttore della superclasse di ***NomeSottoclasse*** passando ***eventualiParametri*** (che possono essere anche diversi dai ***parametri*** del costruttore della sottoclasse)
- Nota: deve essere il primo enunciato del costruttore
- Nota: se non è indicato esplicitamente, viene invocato implicitamente **super( )** senza parametri

Il riferimento *super*

- Il riferimento **`super`** viene usato per riferirsi alla superclasse.
  - Per accedere a variabili, a metodi o a costruttori della superclasse
  - Spesso occorre usare il costruttore dell'antenato per inizializzare i campi di un oggetto figlio, relativi all'antenato, cioè ereditati dalla superclasse

Sovrascrivere un metodo: Esempio

- Proviamo a completare il metodo
  - dobbiamo versare **amount** nel conto, cioè sommarlo a **balance**
  - non possiamo modificare direttamente **balance**, che è una variabile privata in **BankAccount**
  - sappiamo anche che l'unico modo per aggiungere una somma di denaro a **balance** è l'invocazione del metodo **deposit**

```
public class SavingsAccount extends BankAccount
{  ...
   public void deposit(double amount)
   {  withdraw(FEE);
      // invoca deposit della superclasse
      super.deposit(amount);
   }
}
```

Per invocare deposit(amount) di BankAccount si deve usare "super.", altrimenti il metodo diventa ricorsivo con ricorsione infinita.

Ereditarietà singola e multipla

- Java fornisce solo ***ereditarietà singola***
  - La sottoclasse deriva da una sola superclasse
- ***L'ereditarietà multipla*** consente la derivazione di una classe da più classi antenati, e di ereditare i membri di più classi
- L'ereditarietà multipla introduce ambiguità, che devono essere risolte
  - Ad esempio: lo stesso nome di variabile in più classi
- Nella maggior parte dei casi, l'uso di *interfacce* supera i limiti dell'ereditarietà singola senza le complicazioni di quella multipla

Intervallo

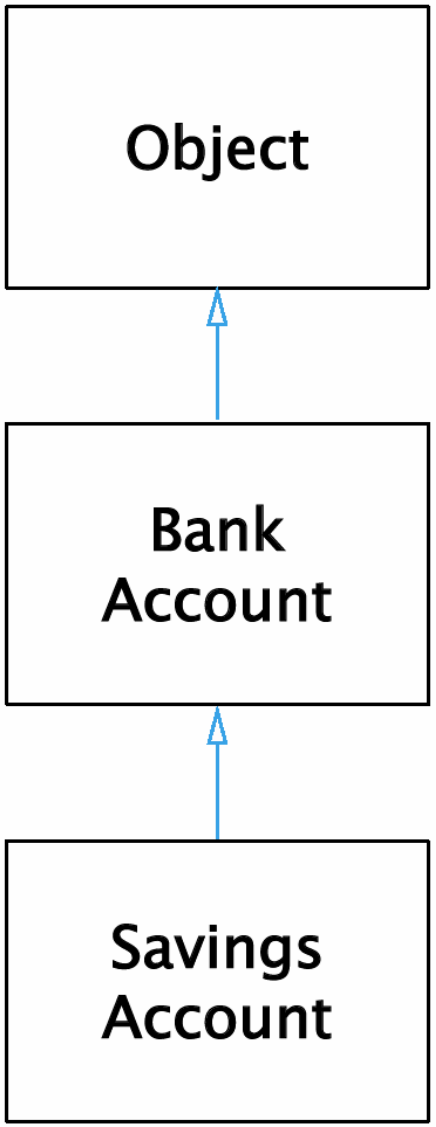
La superclasse universale Object

- In Java, ogni classe che non deriva da nessun'altra deriva ***implicitamente*** dalla ***superclasse universale del linguaggio***, che si chiama **Object**
- Quindi, **SavingsAccount** deriva da **BankAccount**, che a sua volta deriva da **Object**
- **Object** ha alcuni metodi, che vedremo più avanti che quindi sono ereditati da tutte le classi in Java
  - l'ereditarietà avviene anche su più livelli, quindi **SavingsAccount** eredita anche le proprietà di **Object**

La classe Object

Alcune importanti funzioni definite in Object (e quindi ereditate da ogni classe):

- public String toString();
- public boolean equals (Object ob);
- protected Object clone();


Object
Bank Account
Savings Account

Sovrascrivere il metodo toString

Sovrascrivere il metodo toString

- Abbiamo già visto che la classe **Object** è la superclasse universale, cioè tutte le classi definite in Java ne ereditano implicitamente i metodi, tra i quali

```
public String toString()
```

- L'invocazione di questo metodo per qualsiasi oggetto ne restituisce la ***descrizione testuale standard***
  - ***il nome della classe*** seguito dal carattere @ e dall'***indirizzo dell'oggetto in memoria***

```
BankAccount@111f71
```

Sovrascrivere il metodo toString

- Il fatto che tutte le classi siano derivate (anche indirettamente) da **Object** e che **Object** definisca il metodo **toString** consente il funzionamento del metodo **println** di **PrintStream**
  - passando un oggetto di qualsiasi tipo a **System.out.println** si ottiene la visualizzazione della descrizione testuale standard dell'oggetto
  - come funziona **println**?
    - **println** invoca semplicemente **toString** dell'oggetto, e l'invocazione è possibile perché tutte le classi hanno il metodo **toString**, eventualmente ereditato da **Object**

Sovrascrivere il metodo toString

- Se tutte le classi hanno già il metodo **toString**, ereditandolo da **Object**, perché lo dovremmo sovrascrivere, ridefinendolo?

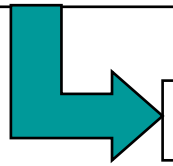
```
BankAccount account = new BankAccount();
System.out.println(account);
```

- Perché in generale la descrizione testuale standard non è particolarmente utile

Sovrascrivere il metodo toString

- Sarebbe molto più comodo, ad esempio per verificare il corretto funzionamento del programma, ottenere una descrizione testuale di **BankAccount** contenente il valore del saldo
  - questa funzionalità non può essere svolta dal metodo **toString** di **Object**, perché chi ha definito **Object** nella libreria standard non aveva alcuna conoscenza della struttura di **BankAccount**
- Bisogna sovrascrivere **toString** in **BankAccount**

```
public String toString()
{  return "BankAccount[balance=" + balance + "]”;
}
```



```
BankAccount[balance=1500]
```

Sovrascrivere il metodo toString

- Il metodo **toString** di una classe viene invocato implicitamente anche quando si concatena un oggetto della classe con una stringa

```
BankAccount acct = new BankAccount();
String s = "Conto " + acct;
```

- Questa concatenazione è sintatticamente corretta, come avevamo già visto per i tipi di dati numerici, e viene interpretata dal compilatore come se fosse stata scritta così

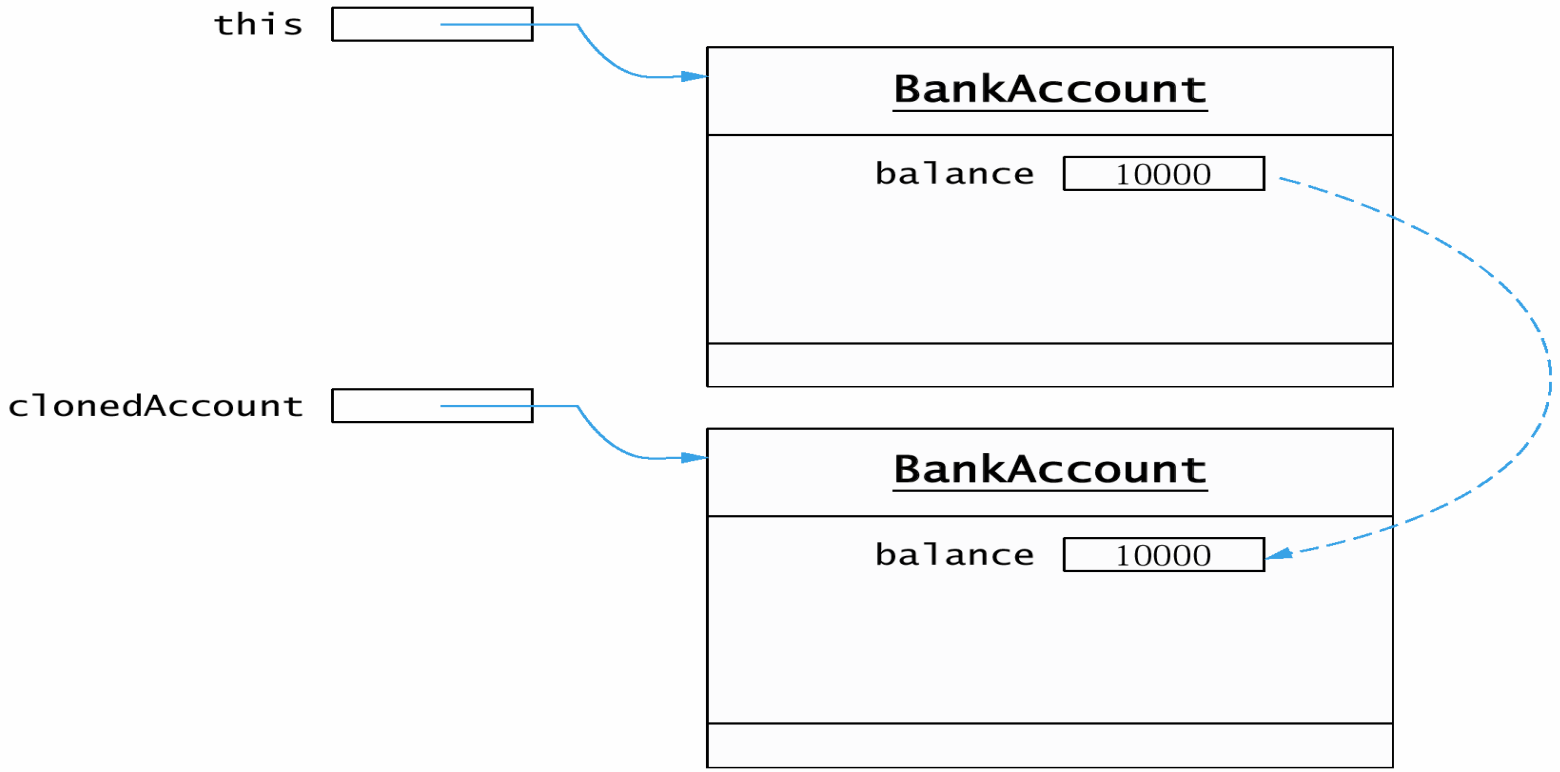
```
BankAccount acct = new BankAccount();
String s = "Conto " + acct.toString();
```

Sovrascrivere sempre toString

- Sovrascrivere il metodo **toString** in tutte le classi che si definiscono è considerato un ottimo stile di programmazione
- Il metodo **toString** di una classe dovrebbe produrre una stringa contenente tutte le informazioni di stato dell'oggetto, cioè
  - il valore di tutte le sue variabili di esemplare
  - il valore di eventuali variabili statiche non costanti della classe
- Questo stile di programmazione è molto utile per il debugging

Clonazione di oggetti

BankAccount clonedAccount = anAccount.clone();



Il metodo Object.clone()

- Crea una copia superficiale
- Non può essere effettivamente invocato a meno che non si implementi l'interfaccia *Cloneable*