Progettare classi

Obiettivi

- Essere in grado di progettare e realizzare semplici classi che generano oggetti
  - Definire le variabili d'istanza (o di esemplare)
  - Definire e realizzare i metodi
  - Definire e realizzare i costruttori
  - Commentare la classe per ottenere la documentazione

Il progetto di BankAccount.java

Progettare la classe

- Dati
  - Gli oggetti (di quasi tutti i tipi) hanno bisogno di ***memorizzare il proprio stato***, cioè ***l'insieme dei valori che descrivono l'oggetto e che determinano gli effetti prodotti dall'invocazione dei metodi dell'oggetto***
    - Quali dati memorizzare per rappresentare lo stato della classe?
    - Quale tipo di dati utilizzo?
    - Che livello di accesso permettere all'utente?
- Metodi
  - Quali metodi definire per:
    - Accedere al valore delle variabili che descrivono lo stato del sistema
    - Modificare il valore delle variabili che descrivono lo stato del sistema
    - Scomporre elaborazioni complesse interne alla classe

Il progetto di BankAccount

Le variabili di esemplare (campi o attributi)

Definire lo stato della classe

- Quali dati memorizzare?
  - Gli oggetti della classe **BankAccount** hanno bisogno di memorizzare ***il valore del saldo*** del conto bancario che rappresentano
  - Come rappresento il saldo?
    - Utilizziamo euro, quindi sarà un valore di tipo ***double***
    - Voglio dare un nome significativo: ***balance***
    - Non voglio che l'utente possa accedere direttamente al valore del saldo, ma che possa usare solo l'interfaccia pubblica
- Le variabili che descrivono lo stato di un oggetto vengono chiamate ***variabili di esemplare*** o anche ***variabili di istanza***, o anche ***instance variables***

Variabili di esemplare

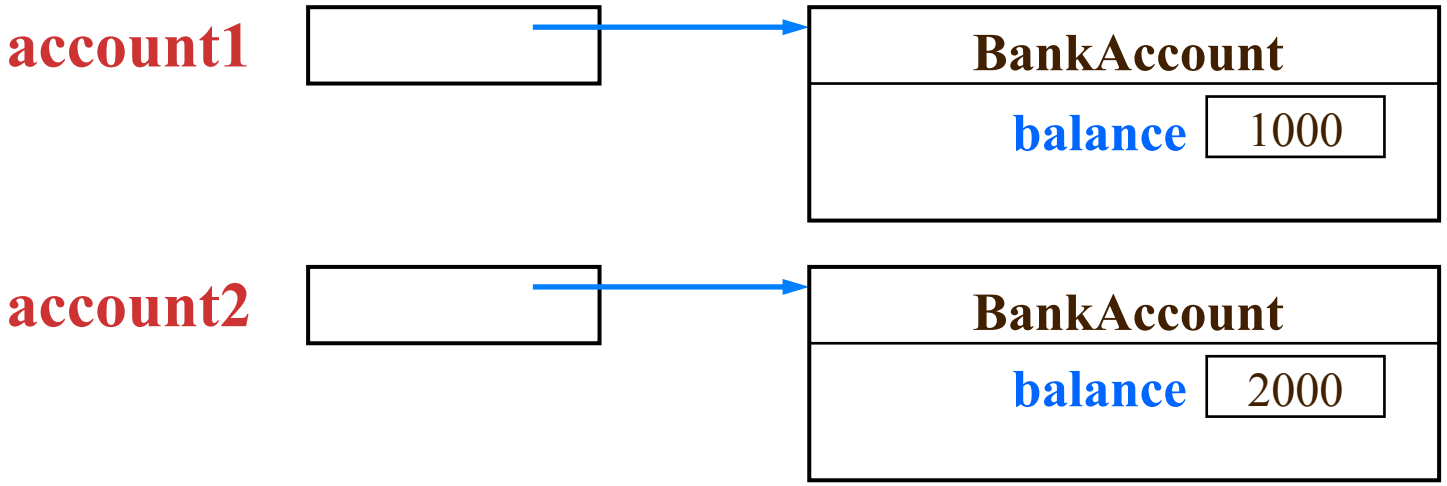
```
public class BankAccount
{
   private double balance; // fuori dai metodi!
   ...
}
```

- La ***dichiarazione*** di una ***variabile di esemplare*** è costituita da
  - uno specificatore di accesso
    - quasi sempre **private**, alcune volte **public**
  - il tipo del dato contenuto nella variabile (**double** nell'esempio)
  - il nome della variabile (**balance**, nell'esempio)
- **Le variabili di esemplare sono definite dentro la classe, ma fuori dai metodi**

Variabili di esemplare

```
public class Tester{
    public static void main(String[] args){
        Bankaccount account1 = new BankAccount(1000);
        Bankaccount account2 = new BankAccount(2000);
        . . .
    }
}
```

- ***Ciascun oggetto*** ("esemplare") della classe ha una ***propria copia*** delle variabili di esemplare

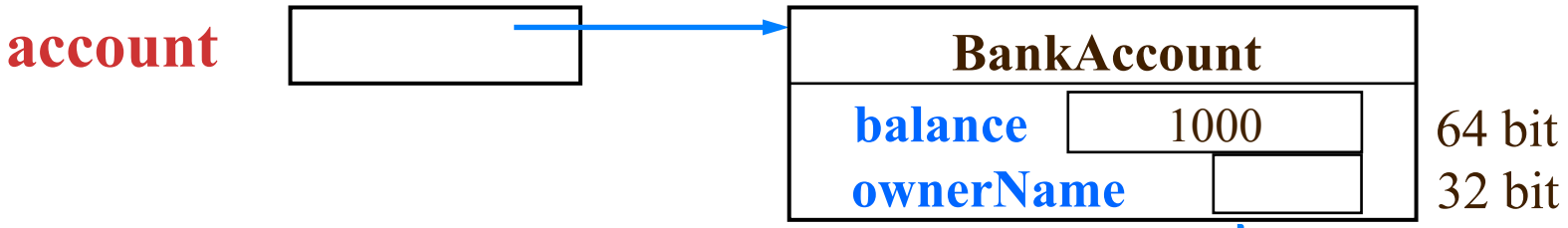


tra tali copie non esiste nessuna relazione: possono essere modificate indipendentemente l'una dall'altra invocando metodi con la variabile oggetto corrispondente

Variabili di esemplare

- Se servono più variabili di esemplare, vengono semplicemente elencate una dopo l'altra

```
public class BankAccount
{  private double balance;
   private String ownerName;
   …
}
```



In memoria, non ci sono "oggetti dentro altri oggetti" ma un oggetto può contenere un riferimento a un altro oggetto!

La variabile di esemplare **ownerName** è una variabile riferimento, che punta a un oggetto di tipo **String**

Dichiarazione di variabili di esemplare

- Sintassi:

```
public class NomeClasse
{   ...
    tipoDiAccesso TipoVariabile nomeVariabile;
    ...
}
```

- Scopo: definire una variabile (di esemplare) ***nomeVariabile*** di tipo ***TipoVariabile***, una cui copia indipendente sia presente in ogni esemplare della classe ***NomeClasse***
  - Di solito le variabili di esemplare sono **`private`**

CODICE
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Information Hiding e incapsulamento

- Le variabili di esemplare sono di solito "private" (*private*)
  - sono nascoste (hidden) al programmatore che utilizza la classe
  - possono essere lette o modificate soltanto mediante l'invocazione di metodi pubblici della classe in cui sono definite
  - questa caratteristica dei linguaggi di programmazione orientati agli oggetti si chiama **incapsulamento**
    - In java è il costrutto ***class*** che realizza l'incapsulamento

Incapsulamento

- Poiché la variabile balance di BankAccount è private, non vi si può accedere da metodi che non siano della classe (errore semantico segnalato dal compilatore)

```
/* codice interno ad un metodo che
   non appartiene a BankAccount */
double b = account.balance; // ERRORE
```

- Si possono usare solo i metodi pubblici!

```
double b = account.getBalance(); // OK
```

Incapsulamento

- Vantaggio fondamentale
  - impedire l'accesso incontrollato allo stato di un oggetto, impedendo così anche che l'oggetto venga (accidentalmente o deliberatamente) posto in uno stato inconsistente
- Esempio:
  - Il progettista della classe BankAccount potrebbe definire (ragionevolmente) che soltanto un saldo non negativo rappresenti uno stato consistente per un conto bancario
    - I metodi della classe saranno implementati in modo che non si verifichino stati inconsistenti

Incapsulamento

- Altri vantaggi
  - Il programmatore che usa una classe non deve preoccuparsi dei dettagli implementativi dei metodi che utilizza
  - Il progettista di una classe può cambiare il codice di un metodo (ad esempio per renderlo più efficiente) senza che il programmatore che e usa la classe (attraverso l'interfaccia) debba modificare il proprio codice
  - In progetti complessi con molte classi, in presenza di un errore relativo ad una particolare variabile d'istanza, posso limitare la ricerca dell'errore alla classe che la contiene

Incapsulamento

- Nel progetto di una classe è buona abitudine prevedere la scrittura di metodi **(pubblici)** per gestire l'accesso alle variabili di esemplare (dichiarate **private**)
- Per ogni variabile avremo bisogno di due metodi:
  - uno per l'accesso in lettura
  - uno per l'accesso in scrittura
- Per convenzione, tali metodi vengono chiamati
  - metodo getNomeVariabile()
  - metodo setNomeVariabile(…)

  dove nomeVariabile è il nome della variabile di esemplare

Incapsulamento

- Dato che il valore di balance può essere modificato **soltanto** invocando i metodi deposit() o withdraw(), il progettista può **impedire che diventi negativo,** magari segnalando una condizione d'errore
  - Se invece fosse possibile assegnare direttamente un valore a balance dall'esterno, ogni sforzo del progettista di **BankAccount** sarebbe vano
  - Si noti che, per questo stesso motivo e anche per realismo, non esiste un metodo setBalance() (mentre esiste getBalance()), dato che il saldo di un conto bancario non può essere impostato ad un valore qualsiasi!

Il progetto di BankAccount

I metodi

Il progetto di BankAccount

- Per manipolare le variabili d'istanza occorre ***realizzare i metodi di esemplare (o d'istanza o non statici)***
  - deposit
  - withdraw
  - getBalance

File
BankAccount.java

```
public class BankAccount
{
    private double balance;

    public void deposit(double amount)
    {  //realizzazione del metodo
    }
    public void withdraw(double amount)
    {  //realizzazione del metodo
    }
    public double getBalance()
    {  //realizzazione del metodo
    }
}
```

Le intestazioni dei metodi

```
public void deposit(double amount)
public double getBalance()
```

- La ***definizione di un metodo*** inizia sempre con la sua ***intestazione*** (o firma, *signature*), composta da
  - uno specificatore di accesso
    - in questo caso **public**, altre volte **private**
  - **NON si mette la parola chiave static!**
  - il tipo del dato restituito dal metodo (**double**, **void**...)
  - il nome del metodo (**deposit**, **withdraw**, **getBalance**)
  - un **elenco di parametri**, eventualmente vuoto, racchiuso tra **parentesi tonde**
    - più parametri sono separati da una virgola;
    - di ogni parametro si indica il tipo e il nome;

Lo specificatore di accesso

- Lo ***specificatore di accesso*** di un metodo indica ***quali altri metodi lo possono invocare***
- Dichiarando un metodo ***public*** si consente la sua invocazione da parte di ***qualsiasi altro metodo presente in qualsiasi altra classe***
  - è comodo per programmi semplici e **con i metodi NON statici *faremo sempre così***, salvo casi eccezionali
- Un metodo che deve essere usato solo per elaborazioni interne alla classe e non invocato da altre classi va dichiarato ***private***
- *Esiste anche lo specificatore* ***protected****… ma lo conosceremo piu' avanti*

Il tipo di dati restituito

- La dichiarazione di un metodo specifica quale sia il tipo di dati restituito dal metodo al termine della sua invocazione
  - ad esempio, **getBalance** restituisce un valore di tipo **double**

```
double b = account.getBalance();
```

- Se un metodo ***non restituisce alcun valore***, si dichiara che restituisce il tipo speciale **void** (assente, non valido…)

```
double b = account.deposit(500); // ERRORE
account.deposit(500);            // OK
```

I metodi di BankAccount

- La realizzazione dei metodi di **BankAccount** è molto semplice
  - lo stato dell'oggetto (il saldo del conto) è memorizzato nella variabile di esemplare **balance**
  - quando si deposita o si preleva una somma di denaro, il saldo del conto si incrementa o si decrementa della stessa somma
  - il metodo **getBalance** restituisce il valore del saldo corrente memorizzato nella variabile **balance**
- Per semplicità, questa prima realizzazione non impedisce che un conto assuma saldo negativo

I metodi di BankAccount

```
public class BankAccount
{  private double balance;

   public void deposit(double amount)
   {  balance = balance + amount;
   }
   public void withdraw(double amount)
   {  balance = balance - amount;
   }
   public double getBalance()
   {  return balance;
   }

}
```

Il progetto è finito!

Gli oggetti di tipo `BankAccount` possono essere creati e funzioneranno come previsto!

L'enunciato return

- Sintassi:

```
return espressione;
```

```
return;
```

- Scopo: terminare l'esecuzione di un metodo, ritornando all'esecuzione sospesa del metodo invocante
  - se è presente una espressione, questa definisce il valore restituito dal metodo e deve essere del tipo dichiarato nella firma del metodo
- Al termine di un metodo con valore restituito di tipo void, viene eseguito un return implicito
  - il compilatore segnala un errore se si termina senza un enunciato return un metodo con un diverso tipo di valore restituito

I parametri dei metodi

```
public void deposit(double amount)
{  balance = balance + amount;
}
```

- Cosa succede quando invochiamo il metodo?

```
account.deposit(500);
```

- L'esecuzione del metodo dipende da ***due valori***
  - il riferimento all'oggetto **account**, che punta all'oggetto (esemplare di **BankAccount**) da elaborare
  - il valore **500**
- Quando viene eseguito il metodo, il suo ***parametro esplicito*** **amount** assume il valore **500**
  - ***esplicito perché compare nella firma del metodo***

I parametri dei metodi

```
public void deposit(double amount)
{  balance = balance + amount;
}
```

- Nel metodo vengono utilizzate due variabili
  - **amount** è il ***parametro esplicito*** del metodo
  - **balance** si riferisce alla ***variabile di esemplare*** **balance** della classe **BankAccount**, ma sappiamo che di tale variabile esiste ***una copia per ogni oggetto***
- Alla variabile **balance** ***di quale oggetto*** si riferisce il metodo?
  - si riferisce alla variabile che appartiene all'***oggetto con cui viene invocato il metodo***, ma come fa?

Il parametro implicito dei metodi

- All'interno di ciascun metodo, il riferimento all'oggetto con il quale è eseguito il metodo si chiama ***parametro implicito*** e si indica con la parola chiave **this**
  - in questo caso, all'interno del metodo **deposit**, **this** assume il valore di **account**

```
account.deposit(500);
```

  - Ogni metodo NON statico ha sempre uno e un solo parametro implicito, dello stesso tipo della classe a cui appartiene il metodo
- Il parametro implicito ***non deve essere dichiarato*** e si chiama sempre **this**

Uso del parametro implicito

- La ***vera sintassi*** del metodo dovrebbe essere

```
public void deposit(double amount)
{   this.balance = this.balance + amount;
}
// this è di tipo BankAccount
```

ma ***nessun programmatore Java scriverebbe così,***

perché Java consente una ***comoda scorciatoia***

- quando in un metodo non statico si fa riferimento a una variabile di esemplare, il compilatore usa ***automaticamente*** un riferimento alla variabile di esemplare dell'oggetto rappresentato dal parametro implicito **this**

Metodi statici (static)

- Non hanno il parametro implicito
  - sono definiti (necessariamente) in una classe, ma ***non hanno accesso alle sue (eventuali) variabili di esemplare*** (perché non hanno **this**…)
    - si usano quando ***non c'è bisogno*** di leggere o scrivere informazioni di stato
    - si invocano con **NomeClasse.nomeMetodo(…)**
  - nome più corretto: "metodo di classe"
    - gli altri: "metodo di esemplare", o "non statico"
  - **Di solito** una classe ha soltanto metodi statici oppure ha soltanto metodi non statici (in questo secondo caso, viene usata come schema progettuale per costruire oggetti), ma si possono definire classi "miste"