

Corso di base JAVA

Mauro Donadeo

mail: mauro.donadeo@gmail.com

Utilizziamo la tastiera



I dati in ingresso

- I programmi visti finora non sono molto utili, visto che **eseguono sempre la stessa elaborazione ad ogni esecuzione**
- Il programma **Coins1** rappresenta sempre il medesimo borsellino...
 - se si vuole che calcoli il valore contenuto in un diverso borsellino, è necessario modificare il codice sorgente (in particolare, le inizializzazioni delle variabili) e compilarlo di nuovo
- I programmi hanno bisogno di **ricevere i dati in ingresso** dell'utente.

- Il modo più semplice e immediato per fornire dati in ingresso ad un programma consiste nell' **utilizzo della tastiera**.
- Abbiamo visto che tutti i programmi Java hanno accesso al proprio *output standard*, tramite l'oggetto **System.out** di tipo **PrintStream**
- Analogamente, l'interprete Java mette a disposizione dei programmi in esecuzione il proprio input standard (**flusso di input**), tramite l'oggetto **System.in** di tipo **InputStream**

La classe Scanner

- Sfortunatamente, la classe **InputStream** non possiede metodi comodi per la ricezione di dati numerici e stringhe
- Per ovviare a questo inconveniente, Java 5.0 ha introdotto la classe **Scanner**
 - Un oggetto di tipo **Scanner** consente di leggere qualsiasi flusso di ingresso
 - Incominceremo ad utilizzarlo per leggere da tastiera utilizzando l'oggetto **System.in**

Usare la classe Scanner

Per leggere dallo standard input bisogna *creare* un oggetto di tipo `Scanner`, usando la sintassi consueta:

```
Scanner in = new Scanner(System.in);
```

- Dato che la classe **Scanner** non fa parte del pacchetto **java.lang**, ma del pacchetto **java.util**, è necessario importare *esplicitamente* la classe all'interno del file java che ne fa uso
- **import** java.util.Scanner;

I metodi `nextInt()` e `nextDouble()`

- Per acquisire un numero intero si può utilizzare il metodo **`nextInt()`**;
 - **`int number = in.nextInt();`**
- Numero in virgola mobile: metodo **`nextDouble()`**;
 - **`double number = in.nextDouble();`**
- Durante l'esecuzione del metodo **il programma si ferma ed attende** l'introduzione dell'input da tastiera, che termina quando l'utente batte il tasto **Invio**

I metodi next e nextLine

Parola

- ovvero una stringa delimitata dai **caratteri di spaziatura**: space, tab, newline, carriage-return
- metodo **String next()**
- **String state = in.next();**

Riga

- ovvero tutta l'intera stringa che viene inserita
- **String city = in.nextLine();**

ESERCIZIO

Modifichiamo il programma Coins1 e per il calcolo dell'ipotenusa, dell'area e perimetro del rettangolo e quello che crea la password affinché acquisiscano i dati da tastiera.

Come progettare le classi.

Progetto di una classe BankAccount

Vogliamo progettare la classe **BankAccount**, che descriva il funzionamento di un conto corrente

Progetto di una classe BankAccount

Vogliamo progettare la classe **BankAccount**, che descriva il funzionamento di un conto corrente

Caratteristiche

- Possibilità di versare denaro;
- Possibilità di prelevare denaro;
- Possibilità di conoscere il saldo attuale

Le operazioni consentite dal comportamento di un oggetto si effettuano mediante invocazione di metodi

Metodi di accesso e modificatori

- **Metodo d'accesso:** accede ad un oggetto e restituisce informazioni senza modificarlo;
 - `length` della classe **String** è metodo di accesso
 - `getX`, `getY`, `getWidth`, `getHeight` della classe **Rectangle** sono metodi di accesso
- **Metodo modificatore:** **altera lo stato** di un oggetto.
 - `translate` della classe **Rectangle** è un metodo modificatore.

Abbiamo detto che per creare un nuovo oggetto di una classe si usa l'operatore **new** seguito dal nome della classe e da una coppia di parentesi tonde

I costruttori

- Nella realizzazione della classe BankAccount bisogna includere il codice per creare un nuovo conto bancario, ad esempio con saldo iniziale a zero
- Per consentire la creazione di un nuovo oggetto di una classe, inizializzandone lo stato, dobbiamo scrivere un nuovo metodo, il **costruttore della classe**

Sintassi

tipoAccesso NomeClasse(TipoParametro nomeParametro,...)

- Lo scopo principale di un costruttore è quello di **inizializzare** un oggetto della classe
- I costruttori, come i metodi, sono solitamente pubblici, per consentire a chiunque di creare oggetti della classe
- I costruttori **non** restituiscono alcun valore.

Invocazione di costruttori

I costruttori si invocano soltanto con l'operatore `new`:

- `new BankAccount();`

`new`

L'operatore **new** riserva la memoria per l'oggetto, mentre il costruttore definisce il suo stato iniziale. Il valore restituito dall'operatore **new** è il riferimento all'oggetto appena creato e inizializzato.

- quasi sempre il valore dell'operatore **new** viene memorizzato in una variabile oggetto.

Più costruttori

Una classe può avere più di un costruttore.

Più costruttori

Una classe può avere più di un costruttore.

```
public BankAccount(){  
    // corpo del costruttore  
    // inizializza il saldo a 0  
}  
public BankAccount(double initialBalance){  
    // corpo del costruttore  
    // inizializza il saldo a initialBalance  
}
```

Notiamo che, se esistono più costruttori in una classe, hanno tutti lo stesso nome, perché devono comunque avere lo stesso nome della classe

- questo fenomeno (più metodi o costruttori con lo stesso nome) è detto sovraccarico del nome (overloading)
- il compilatore decide quale costruttore invocare basandosi sul numero e sul tipo dei parametri forniti nell'invocazione

Definizione di classe

```
tipoAccesso class nomeClasse{  
    costruttori (intestazione e corpo)  
    metodi (intestazione e corpo)  
    variabili (campi) di esemplare  
}
```

Le variabili di esemplare memorizzano lo stato di un oggetto

- La classe bankAccount deve avere un campo di esemplare che permetta di memorizzare il saldo di un oggetto di tipo bankAccount

```
public class BankAccount{  
    //Costruttori  
    public BankAccount(){  
        // corpo del costruttore  
    }  
    public BankAccount(double initialBalance){  
        //corpo del costruttore  
    }  
    //Metodi  
    public void deposit(double amount){  
        //realizzazione del metodo  
    }  
    public void withdraw(double amount){  
        //realizzazione del metodo  
    }  
    public double getBalance(){  
        //realizzazione del metodo  
    }  
    //Campi di esemplare
```

Lo stato di un oggetto

- Gli oggetti hanno tutti bisogno di memorizzare il proprio stato attuale, cioè l'insieme di valori che:
 - **descrivono** l'oggetto;
 - **influenzano** il risultato dell'invocazione dei metodi dell'oggetto.

BankAccount

Nel nostro esempio abbiamo bisogno di memorizzare il valore del **saldo** del conto bancario, che rappresenta lo stato dell'oggetto mediante una **variabile di esemplare**

Sintassi

tipoDiAccesso TipoVariabile nomeVariabile

Ciascun oggetto della classe ha una propria copia delle variabili esemplare. Tra le quali non esiste **alcuna relazione**: possono essere modificate indipendentemente.

Così come i metodi sono di solito **public**, le variabili esemplare sono di solito **private**, in questo modo possono essere modificate **soltanto** da metodi della classe a cui appartengono.

Incapsulamento

Poiché la variabile `balance` **BankAccount** è private, non vi si può accedere da metodi che non siano della classe

```
//codice interno a metodo che non appartiene a ↔  
    BankAccount  
double b = account.balance  
// ERRORE balance has private access in BankAccount
```

Si utilizzano metodi pubblici

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

Il vantaggio fondamentale è quello di **impedire l'accesso incontrollato** allo stato di un oggetto, impedendo così anche che l'oggetto venga posto in uno stato inconsistente.

Dato che il valore di **balance** può essere modificato soltanto invocando metodi **deposit** o **withdraw**, il progettista può impedire che diventi negativo, magari segnalando una **condizione d'errore**

Realizzazione costruttori e metodi.

La realizzazione dei costruttori e dei metodi **BankAccount** è molto semplice:

- lo stato dell'oggetto è memorizzato nella **variabile di esemplare balance**
- i costruttori devono **inizializzare** la variabile balance.
- quando si deposita o si preleva una somma di denaro, il saldo del conto si **incrementa o decrementa** della somma specificata
- il metodo **getBalance** restituisce il valore del saldo corrente memorizzato nella variabile **balance**

I costruttori di BankAccount

```
public class BankAccount{  
    public BankAccount(){  
        balance = 0;  
    }  
    public BankAccount(double initialBalance){  
        balance = initialBalance;  
    }  
    ...  
}
```

Costruttore predefinito

In caso di assenza di un costruttore il compilatore genera **un costruttore predefinito** senza segnalazione di alcun errore.

- è **pubblico** e non richiede parametri
- **inizializza** tutte le variabili di esemplare
 - a zero tutte le variabili di tipo **numerico**
 - a false le variabili di tipo **boolean**
 - al valore speciale **null** le variabili **oggetto**, in modo che tali variabili non si riferiscano a nessun oggetto.

I metodi di BankAccount

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

L'enunciato return

Sintassi

return espressione;

Scopo

Termina l'esecuzione di un metodo ritornando all'esecuzione sospesa dal metodo invocante.

- Se è presente una espressione, questa definisce il valore restituito dal metodo e deve essere del tipo dichiarato dalla firma del metodo.

void

Al termine di un metodo con valore restituito di tipo **void**, viene eseguito un **return implicito**

- Il compilatore segnala **errore** se si termina senza un enunciato **return** un metodo con diverso tipo di valore restituito.

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

Invocazione del metodo

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

L'esecuzione del metodo dipende da due valori

- il **riferimento** all'oggetto account;
- il **valore** 500;

Quando viene eseguito il metodo, il suo **parametro esplicito** **amount** assume il valore 500. Naturalmente farà riferimento alla variabile **balance** che appartiene all'oggetto **account**. **account** è **parametro implicito** del metodo.

Il riferimento null

- Una variabile di un tipo numerico fondamentale contiene sempre un valore valido.
- Una variabile oggetto può invece contenere esplicitamente un **riferimento a nessun oggetto valido** assegnando una variabile il valore **null**, che è una parola chiave del linguaggio
 - **BankAccount account = null**
- in questo caso la variabile è comunque inizializzata

Esempio di riferimento null

String è un oggetto, quindi, può contenere il riferimento **null**

```
String greeting = "Hello";  
String emptyString = ""; // stringa vuota  
String nullString = null; // riferimento null  
int x1 = greeting.length(); // vale 5  
int x2 = emptyString.length(); // vale 0  
// nel caso seguente l'esecuzione del programma  
// termina con un errore  
int x3 = nullString.length(); // errore
```

Una variabile oggetto che contiene un riferimento a null **non si riferisce ad alcun oggetto**. Se viene utilizzata per invocare metodi, l'interprete termina l'esecuzione del programma segnalando un'eccezione di tipo **NullPointerException**.

Usare la classe BankAccount

Senza sapere come sia stata realizzata la classe **BankAccount**, siamo in grado di utilizzarla in un programma.

- apriamo un nuovo conto bancario e depositiamo un po' di denaro.

```
double initialDeposit = 1000;  
BankAccount account = new BankAccount();  
System.out.println("Saldo: " + account.getBalance());  
account.deposit(initialDeposit);  
System.out.println("Saldo: " + account.getBalance());
```

- Trasferiamo denaro da un conto ad un altro

```
double amount = 500;  
account1.withdraw(amount);  
account2.deposit(amount);
```

- calcoliamo e accreditaliamo il 5% di interessi di un conto

```
double rate = 0.05; // interessi del 5%  
double amount = account.getBalance() * rate;  
account.deposit(amount);
```

BankAccount non contiene un metodo **main**

- Compilando `BankAccount.java` si ottiene `BankAccount.class`
- Ma **non** possiamo eseguire la classe `BankAccount.class`

Dobbiamo scrivere una **classe collaudo** (o di test) che contenga un metodo `main` nel quale:

- Costruiamo uno o più oggetti della classe da collaudare
- Invochiamo i metodi della classe per questi oggetti
- Visualizziamo i valori restituiti.

Un programma con più classi

Per scrivere semplici programmi **con più classi** si possono utilizzare due strategie (equivalenti):

- Scrivere **ciascuna classe in un file diverso**, ciascuno avente il nome della classe con estensione `.java`:
 - Tutti i file vanno tenuti nella stessa cartella;
 - tutti file vanno compilati separatamente;
 - solo la classe di collaudo (contente il metodo `main`) va eseguita.
- Scrivere **tutte le classi in un unico file**
 - un file `.java` può contenere una sola classe `public`
 - la classe contenente il metodo `main` deve essere `public`.
 - le altre classi non devono essere `public`
 - il file `.java` deve avere il nome della classe `public`

Riassunto: progettare una classe

- ❶ **Capire** cosa deve fare un oggetto della classe
 - Elenco in un linguaggio naturale delle operazioni possibili;
- ❷ Specificare **l'interfaccia pubblica**
 - Ovvero, definire i metodi tramite le loro intestazioni
- ❸ **Documentare** l'interfaccia pubblica
- ❹ Identificare i **campi di esemplare** a partire dalle intestazioni dei metodi
- ❺ **Realizzare** costruttori e metodi
 - Se avete problemi a realizzare un metodo forse dovete riesaminare i passi precedenti
- ❻ **Collaudare** la classe con un programma di collaudo.

DOCUMENTAZIONE

Documentare l'interfaccia pubblica

- I commenti ai metodi sono **importantissimi** per rendere il codice comprensibile a voi ed agli altri.
- Java ha **delimitatori speciali** per commenti di documentazione

```
/**
Preleva denaro dal conto
@param amount importo da ←
    prelevare
*/
public void withdraw(←
    double amount){
    //corpo del metodo
}
```

- **[at]param nomeparametro**
per descrivere un parametro specifico

```
/**  
Ispeziona saldo attuale  
@return saldo attuale  
*/  
public double getBalance()↵  
{  
    //corpo del metodo  
}
```

- **[at]return** per descrivere il valore restituito