Introduzione a Java
2

Giovanni Pardini
pardinig@di.unipi.it

Dipartimento di Informatica
Università di Pisa

6 dicembre 2009

Sommario

Sommario

Esercizio: `BankAccount`

Vogliamo realizzare una (un tipo) classe `BankAccount` i cui oggetti sono dei semplici conti bancari.

Operazioni richieste

- depositare denaro
- prelevare denaro
- chiedere il saldo corrente

Come si procede?

Definiamo una classe `BankAccount`

- variabili d'istanza per rappresentare lo stato degli oggetti
- metodi d'istanza per modellare le operazioni richieste
- costruttore per creare un nuovo oggetto della classe ed inizializzare le variabili relative

Implementazione


```
1 public class BankAccount {
2   // quantita' di soldi sul conto
3   public double balance;
4
5   // costruttore
6   public BankAccount (double initial) {
7     balance = initial;
8   }
9 }
```

- una variabile d'istanza `balance` di tipo `double`

- Il costruttore imposta il valore della variabile `balance` a quello passato come parametro

Metodi d'istanza

```
1  public class BankAccount {
2    ...
3    public void deposit(double amount) {
4      balance = balance + amount;
5    }
6
7    public void withdraw(double amount) {
8      balance = balance - amount;
9    }
10
11   public double getBalance() {
12     return balance;
13   }
14 }
```

Esempio

```
1 BankAccount p1 = new BankAccount(1000);
2 BankAccount p2 = new BankAccount(2000);
```

Vogliamo trasferire 1000 dal conto di `p1` in quello di `p2`

Esempio

Prima soluzione

```
1 BankAccount p1 = new BankAccount(1000);
2 BankAccount p2 = new BankAccount(2000);
3 p1.withdraw(1000);
4 p2.deposit(1000);
```

Seconda soluzione

```
1 BankAccount p1 = new BankAccount(1000);
2 BankAccount p2 = new BankAccount(2000);
3 p1.balance = p1.balance - 1000;
4 p2.balance = p2.balance + 1000;
```

Esempio: conto di supporto (1)

Aggiungiamo un riferimento ad un conto di supporto, da utilizzare per i prelievi quando il credito non è sufficiente

```
1  public class BankAccount {
2    public double balance;  // saldo corrente
3    public BankAccount contoDiSupporto;
4
5
6    public BankAccount (double initial) {  // costruttore
7      this.balance = initial;
8    }
9
10   public void setContoDiSupporto(BankAccount altroConto) {
11     this.contoDiSupporto = altroConto;
12   }
13   ...
```

Esempio: conto di supporto (2)

```
1  // ritorna true sse riesce a prelevare i soldi richiesti
2  public boolean withdraw(double amount) {
3    if (amount <= balance) { // saldo sufficiente
4        balance -= amount;
5        return true;
6    }
7
8    // saldo insufficiente
9    if (contoDiSupporto == null) // non c'e' conto di supporto
10       return false;
11
12   // prova a prelevare dal conto di supporto
13   double resto = amount - balance;
14   if (contoDiSupporto.withdraw(resto)) {
15       balance = 0;
16       return true;
17   } else {
18       return false;
19   }
20 }
```

Esempio: conto di supporto (3)

```
1  public class BankAccountTest1 {
2    public static void main(String[] args) {
3        BankAccount acc1 = new BankAccount(100);
4        BankAccount altro = new BankAccount(400);
5
6        System.out.println(acc1.withdraw(250)); // false
7        System.out.println(acc1.getBalance());  // 100
8
9        acc1.setContoDiSupporto(altro);
10       System.out.println(acc1.withdraw(250)); // true
11       System.out.println(acc1.getBalance());  // 0
12       System.out.println(altro.getBalance()); // 250
13   }
14 }
```

Utilizzi delle classi

Il meccanismo delle classi è utilizzato in Java per due finalità differenti

1. raggruppare procedure stand-alone
   - utile per la modularità del codice
   - **Esempio**: classe standard `Math` di Java, che fornisce implementazioni di varie funzioni matematiche: `sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, ...
2. dichiarare nuovi tipi di dato
   - **Esempio**: una classe `Vector2D` per rappresentare vettori *(in senso matematico)* su un piano 2D, con operazioni (metodi) come: somma, prodotto vettoriale, prodotto scalare, ecc.
   - I singoli vettori descritti da `Vector2D` sono gli oggetti della classe

Metodi e variabili statici

Variabili e metodi statici sono associati alla classe in cui sono dichiarati

Si usano per:

- memorizzare informazioni comuni tra tutti gli oggetti istanze della classe
- realizzare funzionalità indipendenti dagli oggetti

Variabili statiche: esempio (1)

Aggiungiamo un codice identificativo univoco agli oggetti `BankAccount`

- il codice è descritto da un intero
- una variabile statica `prossimoCodice` che mantiene il valore del primo codice disponibile
- un metodo statico per inizializzare il valore di `prossimoCodice`

Variabili statiche: esempio (2)


```
1 public class BankAccount {
2   public static int prossimoCodice;
3
4   public double balance; // saldo corrente
5   public int codice;
6   public BankAccount (double initial) { ... }
7   ...
```

- variabile statica `prossimoCodice`, contenente il primo codice libero
- variabile d'istanza `codice`, contenente il codice univoco del conto

- C'è un'unica variabile `prossimoCodice`, associata alla classe `BankAccount`

Costruttore


```
1  public class BankAccount {
2    public static int prossimoCodice;
3
4    public double balance; // saldo corrente
5    public int codice;
6
7    public BankAccount (double initial) {
8      balance = initial;
9      codice = prossimoCodice;
10     prossimoCodice++;
11   }
12   ...
```

La variabile statica `prossimoCodice` viene incrementata ogni volta che viene creato un nuovo conto

- mantiene sempre aggiornato il primo valore libero

Variabile statica: modifica

```
1 public class BankAccount {
2   public static int prossimoCodice;
3   ...
4   public  static  void setProssimoCodice(int val) {
5     prossimoCodice = val;
6   }
7   ...
```

Il metodo statico `setProssimoCodice` permette di modificare la variabile statica `prossimoCodice`

Esempio

```
1 BankAccount.setProssimoCodice(10);
2 BankAccount p1 = new BankAccount(1000);
3 BankAccount p2 = new BankAccount(2000);
```

La variabile statica `prossimoCodice` prende inizialmente il valore 10

- il metodo statico `setProssimoCodice` è invocato specificando il nome della classe
- I due oggetti avranno codici diversi e la loro creazione modifica la variabile statica `prossimoCodice`

Conto di supporto: esempio (1)

Aggiungiamo un metodo statico `fairWithdraw` che preleva i soldi dal conto corrente e tutti quelli di supporto, *in modo proporzionale alla disponibilità di ogni singolo conto*

Esempio

- conto 1: saldo = 100
- conto 2: saldo = 400
- conto 3: saldo = 500

Prelevando 200 in modo equo si deve ottenere:

- conto 1: saldo = 80
- conto 2: saldo = 320
- conto 3: saldo = 400

Conto di supporto: esempio (2)

```
1  public class BankAccount {
2    // preleva dal conto corrente e da tutti quelli di supporto
3    // in modo equo, cioe' in proporzione al loro saldo attuale
4    public static boolean fairWithdraw(BankAccount account,
5                                       double amount) {
6        double totalBalance = 0;
7        BankAccount ba = account;
8        while (ba != null) {
9            totalBalance += ba.balance;
10           ba = ba.contoDiSupporto;
11       }
12       if (totalBalance < amount) // credito insufficiente
13           return false;
14
15       double k = amount / totalBalance;
16       ba = account;
17       while (ba != null) {
18           ba.balance -= ba.balance * k;
19           ba = ba.contoDiSupporto;
20       }
21       return true;
22   }
```

Conto di supporto: esempio (3)


```
1  public class BankAccountTest2 {
2    public static void main(String[] args) {
3      BankAccount acc1 = new BankAccount(100);
4      BankAccount acc2 = new BankAccount(400);
5      BankAccount acc3 = new BankAccount(500);
6
7      acc1.setContoDiSupporto(acc2);
8      acc2.setContoDiSupporto(acc3);
9
10     System.out.println(BankAccount.fairWithdraw(acc1, 200));
11                                                    // true
12
13     System.out.println(acc1.getBalance());  // 80
14     System.out.println(acc2.getBalance());  // 320
15     System.out.println(acc3.getBalance());  // 400
16   }
17 }
```

Sommario

Terminologia

- **Classe**, rappresenta un tipo di dato
- **Oggetti** o **istanze** della classe *(class instances)*, gli elementi appartenenti ad una classe

La definizione di una classe contiene le definizioni dei membri della classe *(class members)*:

- **variabili** o **campi** della classe *(class fields)*, d'istanza e statiche
- **metodi**, d'istanza e statici

Sommario

Specificatori di accesso

L'ambito di visibilità (*scope*) del nome di un elemento (classe, variabile, metodo) indica le parti del programma in cui il nome riferisce quell'elemento

- per i metodi, l'ambiente locale è gestito dinamicamente *(ambiente locale dinamico)*

Anche se un nome è visibile, l'accesso all'elemento riferito *può* non essere consentito

- si usano gli specificatori di accesso, particolari keyword che specificano dove l'accesso ad un elemento è consentito

Specificatori di accesso *(access specifiers)*

- `public`: l'accesso è consentito ovunque
- `private`: l'accesso è consentito solo all'interno della classe
- `protected`
- *(nessuna keyword) (package access)*

Specificatori di accesso: esempio (1)

Rendiamo le variabili di `BankAccount` private alla classe

```
1  public class BankAccount {
2    private static int prossimoCodice;
3
4    private double balance; // saldo corrente
5    private int codice;
6
7    public  static  void setProssimoCodice(int val) {
8      prossimoCodice = val;
9    }
10
11   public BankAccount (double initial) {
12     balance = initial;
13     codice = prossimoCodice;
14     prossimoCodice++;
15   }
16   ...
```

Tutte le variabili sono `private`, il resto del codice resta inalterato

Specificatori di accesso: esempio (2)

```
1 public class BankAccount {
2   ...
3   private double balance; // saldo corrente
4   ...
5 }
```

```
1 public class MainClass {
2   public static void main(String[] args) {
3     BankAccount p1 = new BankAccount(1000);
4     p1.balance = 0;  // errore: balance non accessibile
5     p1.withdraw(1000);
```

- L'accesso alla variabile d'istanza `balance` di `BankAccount` non è consentito

Package

Java mette a disposizione il concetto di package, quale collezione di classi

Ci sono altri specificatori di accesso, che permettono di definire con maggiore granularità il livello di accesso agli elementi

- **`private`**: l'accesso è consentito solo all'interno della classe di definizione
- *(nessuna keyword)* (package access): l'accesso è consentito all'interno del package
- **`protected`**: l'accesso è consentito all'interno del package e nelle sottoclassi
- **`public`**: l'accesso è consentito ovunque

- Il significato esatto degli specificatori di accesso dipende dall'elemento a cui sono applicati

Incapsulamento

L'incapsulamento è una utile proprietà degli oggetti, per i quali si ha una separazione tra

- l'interfaccia (i metodi) fornita verso l'esterno
- l'implementazione interna, mantenuta nascosta

Vantaggi

- l'implementazione degli oggetti non è visibile dall'esterno
- se i metodi sono corretti, posso garantire che lo stato rimanga sempre "consistente"
- posso cambiare l'implementazione senza che chi usa l'oggetto debba essere modificato

- L'incapsulamento si ottiene utilizzando:
  - variabili d'istanza private
  - metodi d'istanza per accedere in lettura e scrittura alle variabili private

Incapsulamento: esempio (1)


```
1 BankAccount p1 = new BankAccount(1000);
2 BankAccount p2 = new BankAccount(2000);
3 p1.balance = 0;
4 p2.balance = balance + 1000;
```

Il codice accede allo stato interno degli oggetti

- se cambia l'implementazione, il codice sopra deve essere modicato
- non si può garantire, ad esempio, che i conti non vadano in rosso

- Meglio manipolare lo stato degli oggetti *solo* tramite i metodi d'istanza

Incapsulamento: esempio (2)

```
1 BankAccount p1 = new BankAccount(1000);
2 BankAccount p2 = new BankAccount(2000);
3 p1.withdraw(1000);
4 p2.deposit(1000);
```

- Il metodo `withdraw` può garantire che il conto non vada in rosso
  - lanciando una eccezione e facendo fallire l'operazione, se il saldo non è sufficiente

Sommario

Overloading dei metodi

In una classe possono essere definiti più metodi (o costruttori) con lo stesso nome, ma con parametri diversi

- deve essere diverso il numero dei parametri e/o il loro tipo
- il compilatore sceglie il metodo da invocare in base al tipo dei parametri attuali

Esempio

Aggiungiamo due metodi alla classe `Vector2D`, che sommano al vettore corrente un altro vettore

- il secondo vettore può essere specificato in due modi

```
// somma un vettore specificato tramite le coordinate
public void somma (int a, int b) { ... }

// somma un altro oggetto vettore
public void somma (Vector2D v1) { ... }
```

Overloading dei metodi: esempio (1)

```
1  public class Vector2D {
2    private int x;
3    private int y;
4    public void somma (int a, int b) {
5      x += a;
6      y += b;
7    }
8    public void somma (Vector2D v1) {
9      x += v1.x;
10     y += v1.y;
11   }
12 }
```

- Due versioni del metodo somma, che modifica l'oggetto attuale aggiungendo i valori passati

Overloading dei metodi: esempio (2)

```
1  public class Vector2D {
2    private int x;
3    private int y;
4    public void somma (int a, int b) {
5      x += a;
6      y += b;
7    }
8    public void somma (Vector2D v1) {
9      x += v1.x;
10     y += v1.y;
11   }
12 }
```

Esempio

```
1 Vector2D v = new Vector2D(4, 6);
2 v.somma(3,5);                  // primo metodo
3 v.somma(new Vector2D(3,5)); // secondo metodo
```

Overloading dei costruttori

Come per i metodi, l'overloading è possibile anche per i costruttori

Esempio

Aggiungiamo alla classe `Vector2D` un costruttore senza parametri, che crea il vettore nullo (con entrambe le componenti 0)

```
1  public class Vector2D {
2    public int x;
3    public int y;
4
5    // costruttore
6    public Vector2D () {
7      x = 0;
8      y = 0;
9    }
10
11   ...
```

Overloading dei costruttori: esempio

```
1  public class Vector2D {
2    public int x;
3    public int y;
4
5    // costruttore
6    public Vector2D () {
7      x = 0;
8      y = 0;
9    }
10
11   //costruttore
12   public Vector2D (int initx, int inity) {
13     x = initx;
14     y = inity;
15   }
16   ...
```

Esempio

```
1  Vector v1 = new Vector();
2  Vector v2 = new Vector(0,0);
```

Invocare un altro costruttore (1)

Quando si ha overloading dei costruttori, è possibile, da un costruttore, invocarne un altro

- per invocare un altro costruttore si utilizza un comando

```
this( <parametri> );
```

Esempio

```
1  public class Vector2D {
2    public int x;
3    public int y;
4
5    public Vector2D () {
6      this(0,0); // invocazione dell'altro costruttore
7    }
8
9    public Vector2D (int x, int y) {
10     this.x = x;
11     this.y = y;
12   }
13   ...
```

Invocare un altro costruttore (2)

Il comando di invocazione di un altro costruttore può apparire solo come primo comando nel corpo di un costruttore

Esempio

```
1  public class Vector2D {
2    public int x;
3    public int y;
4
5    public Vector2D (int k) {
6      x = k;
7      y = k;
8      this(k,0); // chiamata non consentita in questo punto
9      ...  // altri comandi
10   }
11
12   public Vector2D (int x, int y) {
13     this.x = x;
14     this.y = y;
15   }
16   ...
```

Sommario

Ereditarietà

L'ereditarietà permette di *estendere* classi già definite

- ovvero di definire sottotipi di tipi già definiti
- è utile sia per il riutilizzo del codice, sia per lo sviluppo incrementale di programmi

L'ereditarietà è sfruttata per

- estendere e potenziare classi già esistenti
- fattorizzare informazioni comuni a più classi

- Introduciamo i meccanismi principali

Esempio: classe Persona (1)

```
1  public class Persona {
2    String nome;
3    String indirizzo;
4
5    public Persona() {
6      this.nome = "";
7      this.indirizzo = "";
8    }
9
10   public Persona(String nome, String indirizzo) {
11     this.nome = nome;
12     this.indirizzo = indirizzo;
13   }
14
15   public String getNome() {
16     return nome;
17   }
18
19   public String getIndirizzo() {
20     return indirizzo;
21   }
22   ...
```

Esempio: classe Persona (2)

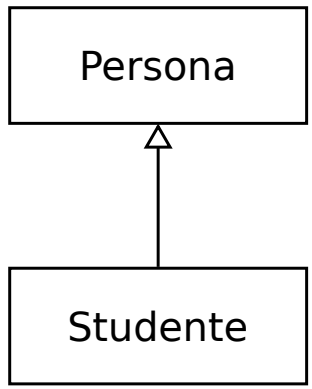
```
1  public class Persona {
2
3    ...
4
5    public void visualizza() {
6      System.out.println("Nome: " + nome);
7      System.out.println("Indirizzo: " + indirizzo);
8    }
9
10   public boolean omonimo(Persona p) {
11     return this.nome.equals(p.nome);
12   }
13 }
```

Esempio: sottoclasse `Studente` (1)

Definiamo una classe `Studente` che rappresenta gli studenti iscritti ad un corso di laurea.

- estendiamo la classe `Persona`
- agli studenti associamo una *matricola* e un *piano di studio*

- Ogni `Studente` **è una** *(is-a)* `Persona`



Esempio: sottoclasse Studente (2)

```
1  public class Studente extends Persona {
2    public static int nextMatricola = 1;
3
4    public int matricola;
5    public String pianoDiStudio;
6
7    public Studente(String nome, String indirizzo) {
8      this.nome = nome;
9      this.indirizzo = indirizzo;
10     this.pianoDiStudio = "";
11     this.matricola = nextMatricola;
12     nextMatricola++;
13   }
14
15   public String getPdS() {
16     return pianoDiStudio;
17   }
18
19   public void setPdS(String nuovoPdS) {
20     pianoDiStudio = nuovoPdS;
21   }
22 }
```

Terminologia

La parola chiave `extends` indica che

- `Studente` è sottoclasse (o *classe derivata*) di `Persona`
- `Persona` è superclasse (o *classe genitrice*) di `Studente`

- Analogamente per i tipi
  - `Studente` è un sottotipo di `Persona`
  - `Persona` è un supertipo di `Studente`

Ereditarietà: variabili e metodi

Se `B` è una sottoclasse di (estende) `A`

- tutti i membri (variabili/metodi) della superclasse A sono definiti automaticamente anche per B

- I campi/metodi della superclasse sono sempre ereditati nella sottoclasse, ma possono non essere accessibili
  - dipende dal livello di accesso consentito al campo/metodo (`private`, *default*, `protected`, `public`)

Esempio (1)

Un oggetto di tipo `Studente` ha le seguenti variabili di istanza:

- `nome` e `indirizzo`, ereditate da `Persona`
- `matricola` e `pianoDiStudio`, definite nella sottoclasse `Studente`

- Come per le variabili, anche tutti i metodi d'istanza della superclasse `Persona` sono presenti in `Studente`
  - ma attenzione al caso particolare di metodi ridefiniti (*overriding*)

Esempio (2)

```
1 Studente p = new Studente("Mario", "Lucca");
2 String s1 = p.getIndirizzo(); // metodo superclasse
3 String s2 = p.getPds();       // metodo sottoclasse
4 String s3 = p.nome;          // variabile superclasse
5 int x = p.matricola;         // variabile sottoclasse
```

Creazione di oggetti

Se `B` è una sottoclasse di `A`, all'atto della creazione di un oggetto di `B` devono essere inizializzate anche tutte le variabili d'istanza della superclasse `A`

- deve essere invocato un costruttore di `A`

Alla creazione di un oggetto della sottoclasse `B`, viene eseguito *prima* il costruttore della superclasse `A`, *poi* quello della sottoclasse `B`

- un costruttore della superclasse viene **sempre** invocato

Costruttori: esempio


```
1 public class Studente extends Persona {
2   public Studente(String nome, String indirizzo) {
3     this.nome = nome;
4     this.indirizzo = indirizzo;
5     ...
6   }
```

Alla creazione di un oggetto Studente, viene invocato automaticamente il costruttore senza parametri di Persona

```
1 public class Persona {
2   String nome;
3   String indirizzo;
4
5   public Persona() {
6     this.nome = "";
7     this.indirizzo = "";
8   }
9 }
```

Costruttori ereditati

Se la superclasse `A` non definisce (anche implicitamente) un costruttore senza parametri, nella sottoclasse `B` deve essere invocato esplicitamente un costruttore di `A`

- si usa il seguente comando come prima istruzione del costruttore della sottoclasse `B`

```
super ( <parametri attuali> );
```

- In generale, il comando `super(...)` può essere usato per invocare un costruttore diverso da quello senza parametri

Costruttori ereditati: esempio

```
1   public Studente(String nome, String indirizzo) {
2     this.nome = nome;           // inizializzazione diretta
3     this.indirizzo = indirizzo; // variabili della superclasse
4
5     this.pianoDiStudio = "";
6     this.matricola = nextMatricola;
7     nextMatricola++;
8   }
```

```
1   public Studente(String nome, String indirizzo) {
2     super (nome, indirizzo); // chiamata esplicita al
3                              // costruttore della superclasse
4
5     this.pianoDiStudio = "";
6     this.matricola = nextMatricola;
7     nextMatricola++;
8   }
```

- Le due versioni sono equivalenti

Overriding

In alcuni casi i metodi ereditati dalla superclasse possono non essere adatti per la sottoclasse

Una sottoclasse può sovrascrivere (*override*) un metodo della superclasse

- definendo un metodo con lo stesso nome e la *stessa* signature (firma)
- la signature si un metodo include: numero e tipo dei parametri, tipo di ritorno

- Sugli oggetti della sottoclasse viene utilizzato il metodo riscritto (più specifico)

Overriding: esempio (1)

Invocando il metodo `visualizza()` su un'istanza di `Studente`, viene eseguito il metodo ereditato da `Persona`

```
1 public class Persona {
2   ...
3   public void visualizza() {
4     System.out.println("Nome: " + nome);
5     System.out.println("Indirizzo: " + indirizzo);
6   }
```

- Vengono stampati solo i valori di `nome` e `indirizzo`

Overriding: esempio (2)

Sovrascriviamo il metodo `visualizza()` in `Studente` per stampare anche la matricola ed il piano di studio

```
1 public class Studente extends Persona {
2   ...
3   public void visualizza() {
4     System.out.println("Nome: " + nome);
5     System.out.println("Indirizzo: " + indirizzo);
6     System.out.println("Matricola: " + matricola);
7     System.out.println("Piano di studio: " + pianoDiStudio);
8   }
9 }
```

- L'invocazione del metodo `visualizza()` su un oggetto `Studente` esegue il metodo riscritto

Overriding: operatore `super`

In una sottoclasse, nel caso di un metodo sovrascritto, è possibile invocare il metodo originale della superclasse con un comando

```
super.<nome metodo> ( <parametri attuali> );
```

Esempio

```
1 public class Studente extends Persona {
2   ...
3   public void visualizza() {
4     super.visualizza(); // chiamata al metodo originale
5     System.out.println("Matricola: " + matricola);
6     System.out.println("Piano di studio: " + pianoDiStudio);
7   }
8 }
```

- Stesso comportamento del codice precedente

Regole di accesso

A seconda del livello di accesso specificato con cui sono dichiarati, i campi/metodi di una classe possono non essere accessibili nelle sottoclassi

- `public`, l'accesso è sempre consentito
- `protected`, l'accesso è consentito all'interno del package e nelle sottoclassi (anche definite in altri package)
- *nessuno (default)*, l'accesso è consentito solo all'interno del package
- `private`, l'accesso è consentito solo all'interno della classe di definizione
  - quindi neache nelle sottoclassi

Regole di accesso: esempio (1)

```
1  public class Persona {
2    private String nome;
3    private String indirizzo;
4    ...
```

```
1  public class Studente extends Persona {
2    public int matricola;
3    public String pianoDiStudio;
4
5    public Studente(String nome, String indirizzo) {
6      this.nome = nome;           // this.nome non visibile
7      this.indirizzo = indirizzo; // this.indirizzo non visibile
8
9      this.pianoDiStudio = "";
10     ...
11   }
```

- Le variabili d'istanza private `nome` e `indirizzo` di `Persona` non sono accessibili dalle sottoclassi

Regole di accesso: esempio (2)

```java
public class Persona {
  private String nome;
  private String indirizzo;
  ...
```

```java
public class Studente extends Persona {
  ...
  public void visualizza() {
    System.out.println("Nome: " + nome); // nome non accessibile
    System.out.println(
        "Indirizzo: " + indirizzo); // indirizzo non accessibile
    System.out.println("Matricola: " + matricola);
    System.out.println("Piano di studio: " + pianoDiStudio);
  }
}
```

- Per evitare il problema, possiamo limitarci ad utilizzare i metodi ereditati

Regole di accesso: esempio (3)

```
1 public class Studente extends Persona {
2   ...
3   public void visualizza() {
4     System.out.println("Nome: " + getNome());            // ok
5     System.out.println("Indirizzo: " + getIndirizzo());  // ok
6     System.out.println("Matricola: " + matricola);
7     System.out.println("Piano di studio: " + pianoDiStudio);
8   }
```

- Per il costruttore, invochiamo esplicitamente il costruttore della superclasse (già visto)

```
1   public Studente(String nome, String indirizzo) {
2     super (nome, indirizzo); // chiamata esplicita al
3                              // costruttore della superclasse
4
5     this.pianoDiStudio = "";
6     this.matricola = nextMatricola;
7     nextMatricola++;
8   }
```

Sommario I