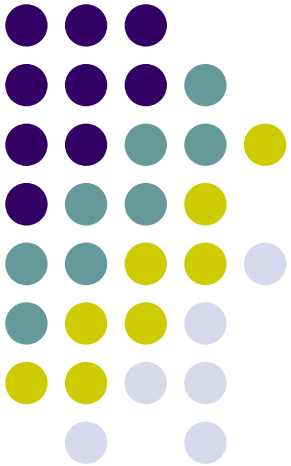
Università di Napoli Federico II - Dip. di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Corso di Laurea in Ingegneria Informatica

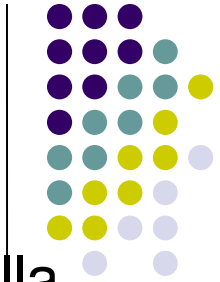
DIETI

Corso di Programmazione

Classi: aspetti avanzati



Metodi e accesso ai membri



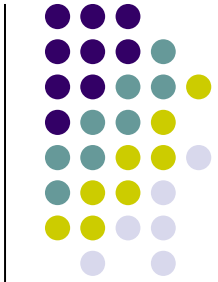
- La nostra prima applicazione Java utilizza più oggetti della classe Counter
- Ad esempio invoca il metodo get_value() sull'oggetto ***c*** e il metodo set_max(…) sull'oggetto ***d***
- Ma se noi guardiamo l'implementazione dei metodi, al loro interno non c'è nulla che dica esplicitamente su quale oggetto il metodo deve essere eseguito!
- Come fa il metodo a riferirsi ai membri dell'oggetto su cui è invocato???

```
public long get_value() {return value;}
public void set_max(long m) {max=m;}
```

```
c.get_value()
d.set_max(20)
```

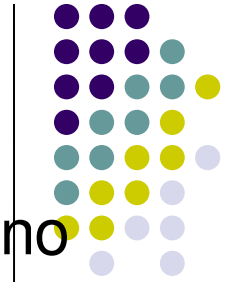
Come faccio a sapere all'interno del metodo se la varaibili di istanza value e max sono quelle di c piuttosto che quelle di d?

Il riferimento *this*



- Nella definizione di TUTTI i metodi NON STATICI di una classe il compilatore introduce un riferimento *nascosto* che è il riferimento ***all'oggetto corrente***, quello sul quale il metodo viene applicato
- Il nome di tale riferimento è *this* (ed è ovviamente un nome riservato del linguaggio)
- All'atto della chiamata del metodo, quando è noto l'oggetto su cui il metodo è invocato, a *this* viene assegnato il riferimento all'oggetto corrente
- *this* ovviamente non può essere modificato all'interno del metodo ma solo usato

Uso implicito di *this*

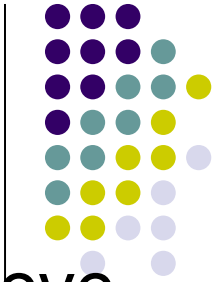


- I metodi quindi riescono a riferirsi all'oggetto su cui sono applicati in esecuzione perché in realtà nel codice viene usato il riferimento *implicito this*
- *Nella maggior parte dei casi this è usato implicitamente, cioè **non è necessario utilizzarlo esplicitamente** per accedere alle variabili di istanza e invocare metodi della classe*
- Nel codice della classe Counter è COME SE fosse scritto:

```
public long get_value() {return this.value;}
public void set_max(long m) {this.max=m;}
```

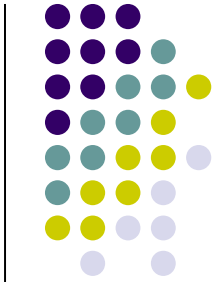
- Quando il metodo vene invocato, ad esempio sull'oggetto *c*, *this* viene inizializzato come riferimento a *c e il metodo lavora quindi direttamente sull'oggetto c tramite this.*

Uso esplicito di *this*



- Il più delle volte quindi il programmatore non deve usare ***this*** esplicitamente
- Si possono determinare però alcune circostanze in cui l'uso esplicito di ***this*** diviene necessario:
  - In generale, per riferirsi a tutto l'oggetto corrente
  - All'interno di un costruttore per invocare un altro costruttore della classe
  - Per disambiguare nel caso si usino variabili locali che hanno lo stesso nome di variabili di istanza (cosa comunque assolutamente sconsigliata)
- Approfondiremo l'utilizzo esplicito di this quando si renderà necessario

Copia di oggetti (1/4)



- In Java come abbiamo visto gli oggetti si utilizzano attraverso riferimenti, che si comportano come puntatori all'oggetto
- Pertanto, dati ad esempio:

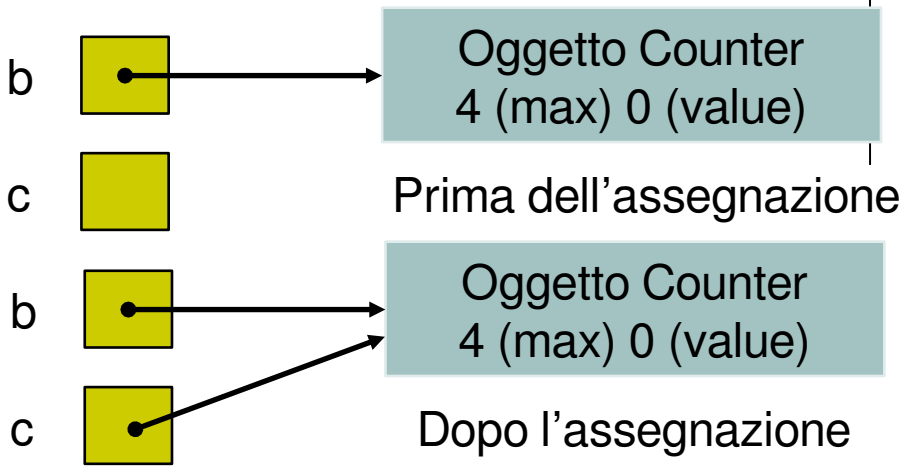
  Counter b=new Counter(4);

  Counter c;
- l'assegnazione: c=b NON effettua una copia tra oggetti ma una copia tra riferimenti
- Se non gestita correttamente quindi la copia può creare situazioni gravi di errore...

Effetti della copia tra riferimenti

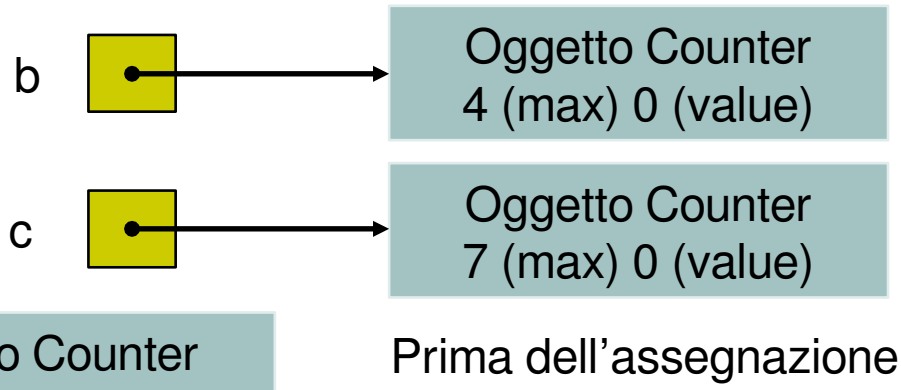
```
Counter b=new Counter(4);
Counter c;
c=b;
```

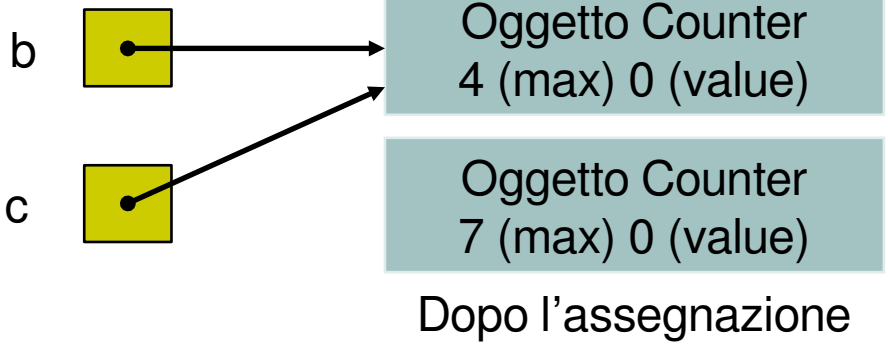
caso1




```
Counter b=new Counter(4);
Counter c=new Counter(7);
c=b;
```

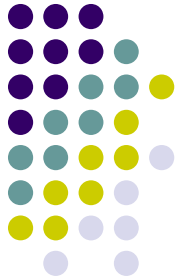
caso2





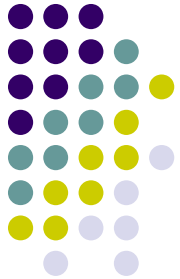
Oggetto orfano!
Verrà deallocato
dal garbage collector

Copia di oggetti (3/4)



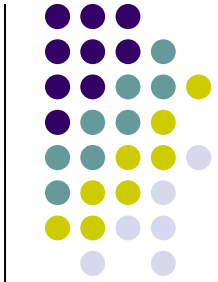
- La copia tra oggetti avviene in due casi:
  - Quando un nuovo oggetto viene istanziato inizializzandolo con lo stato di un altro oggetto della sua classe già esistente (costruttore di copia)
  - Quando lo stato di un oggetto già esistente sovrascrive lo stato di un altro oggetto già esistente della sua classe (assegnazione)
- Abbiamo già visto il caso del costruttore, nella precedente lezione sulle classi, quel costruttore copia lo stato variabile per variabile nel nuovo oggetto (riferito attraverso this)

Copia di oggetti (4/4)



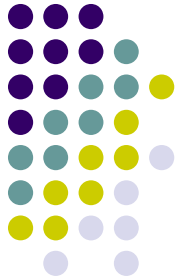
- Nel caso dell'assegnazione come abbiamo capito, NON è possibile utilizzare =
- Per creare davvero una copia si usa tipicamente il metodo **`clone`**, implementato in molte classi di libreria.
- Lo studio e l'utilizzo del metodo clone verrà trattato dopo le lezioni sulla ereditarietà

Confronto tra oggetti



- Lo stesso problema che si presenta per la copia si pone per il confronto tra oggetti
- L'istruzione: b==c NON confronta lo stato di due oggetti ma il valore di due riferimenti!!!
  - Due riferimenti sono uguali se puntano allo stesso oggetto e sono diversi altrimenti
- Per confrontate lo stato degli oggetti Java fornisce il metodo (**`equals`**) di cui è necessario fare l'oveloading per personalizzare il criterio di uguaglianza tra gli oggetti da caso a caso.

Classe Counter: *equals*

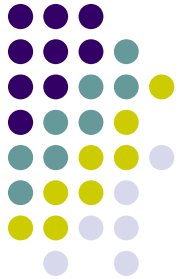


- Facciamo l'overload del metodo ***equals*** per la classe Counter.
- Il metodo deve implementare l'operatore binario == quindi deve operare su due oggetti, l'operando di sinistra è il riferimento all'oggetto corrente al quale si applica il metodo, l'operando di destra è il riferimento all'altro oggetto, che viene passato come parametro al metodo
- Il metodo confronta il valore delle variabili di istanza dell'oggetto corrente (quello implicitamente riferito da ***this***) con il valore delle variabili di istanza dell'altro oggetto
- Il metodo è un predicato, ritorna un valore booleano che sarà vero se se lo stato dei due oggetti è uguale, falso altrimenti

```
public boolean equals(Counter C) {
        return (max==C.max && value==C.value);
   }
```

```
if(c.equals(d)) System.out.print("gli oggetti riferiti da c e d sono uguali \n");
   else System.out.print("gli oggetti riferiti da c e d sono diversi \n");
if(c==d) System.out.print("c e d sono uguali, quindi puntano allo stesso oggetto \n");
  else System.out.print("c e d sono diversi, quindi puntano a oggetti diversi \n");
```

Classe Counter: not_*equal*



- E se volessimo implementare un metodo che realizza l'operatore != (diverso) per gli oggetti Counter?
- Abbiamo due possibilità:
  - Lo implementiamo «from scratch»... scrivendo il codice che confronta le singole variabili di istanza
  - Usiamo ***equals*** (se due oggetti non sono uguali sono diversi...)

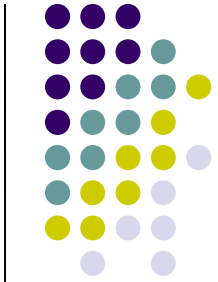
Counter: not_equal

- Implementiamo il metodo ***not_equal*** utilizzando ***equals*** già a nostra disposizione
- Anche ***not_equal*** realizza un operatore binario, quindi si applica ad un riferimento all'oggetto corrente, e riceve il riferimento all'operando di destra come parametro
- Per negare il risultato di ***equal*** dobbiamo applicare al valore ritornato l'operatore **not** che è **!** in Java (e anche in C/C++), in questo modo ***not_equal*** negherà il risultato di ***equal***
- ***Ma a chi dobbiamo applicare equals?***

```
public boolean not_equal(Counter C) {
        return !(this.equals(C));
    }
```

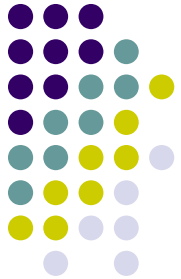
In realtà anche in questo caso ***this*** può essere implicito…

Membri *static:* variabili di classe



- In alcuni casi è utile che gli oggetti di una stessa classe possano accedere ad una area di memoria condivisa per accedere e modificare informazioni che devono essere condivise da tutti gli oggetti della classe
  - Questa area di memoria quindi farà parte della specifica della classe ma la sua creazione deve essere indipendente dagli oggetti della classe
  - Essa viene allocata al via dell'esecuzione dell'applicazione anche se non esistono ancora oggetti della classe
- In Java è possibile introdurre in una classe delle variabili static, che NON sono variabili di istanza perché NON sono diverse per ogni oggetto, ma allocate in area dati statici (quindi né in area heap né in area stack)
- Le variabili ***static*** sono anche dette **variabili di classe** (in contrapposizione alle variabili d'istanza, che sono appunto associate a specifici oggetti).

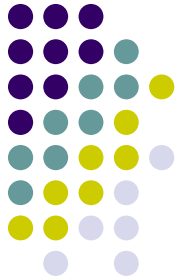
Membri *static*: variabili di classe



- Le variabili di classe vengono create dalla JVM quando viene eseguita la prima istruzione che contiene un riferimento a quella classe
- Tutte le variabili static di una classe vengono inizializzate al valore di default prima che un qualunque altra variabile static della classe venga utilizzata, e prima che qualunque metodo della classe venga utilizzato.
- *E' possibile inizializzare una variabile di classe ad un valore diverso da quello di default* direttamente quando viene dichiarata nella specifica della classe
- Le variabili di classe possono essere sia pubbliche che private.
- Le variabili di classe pubbliche sono accessibili direttamente agli utenti usando il nome della classe e la notazione punto .

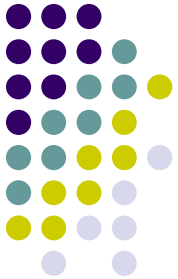
nome_classe.nome_variabile_statica

Membri *static*: metodi statici



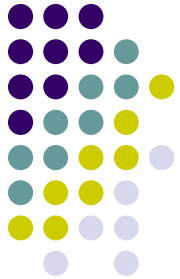
- Le variabili di classe private possono essere utilizzate dagli utenti della classe solo attraverso appositi metodi statici
- I metodi dichiarati *static* possono essere invocati anche in assenza di oggetti della classe e quindi non accedono all'oggetto corrente attraverso il riferimento this.
- I metodi statici possono essere dichiarati anche in assenza di variabili di classe, in questo caso non lavorano su oggetti
- Possono essere chiamati:
  - utilizzando il nome della classe e la notazione punto
  - applicandoli ad un qualunque riferimento ad oggetto della classe esistente

Counter: contatore di istanze



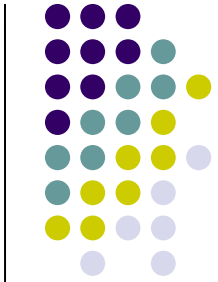
- Supponiamo che si voglia sapere ad ogni istante quanti contatori sono stati creati fino a quel momento.
- Questo significa che tutte le volte che viene chiamato un costruttore deve essere incrementata una variabile statica che tiene il conto del numero di oggetti creati
- Tutti i contatori avranno accesso a questa informazione e potranno modificarla (in questo caso attraverso i costruttori), inoltre deve essere possibile per l'utente interrogare questa variabile statica (attraverso un metodo statico)

Membri static: osservazioni



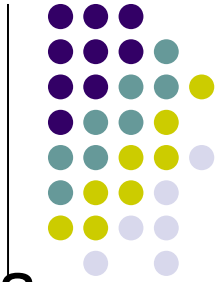
- In generale, è bene limitare il più possibile tutto quanto è **`static`**.
- I metodi statici vanno introdotti solo quando è necessario (tipicamente: il **`main`**, e metodi che agiscono su numeri, che non sono oggetti)
- Le variabili statiche vanno usate con cautela perché a tutti gli effetti sono variabili globali a livello di classe

Variabili di istanza costanti



- Può essere utile poter dichiarare variabili di istanza che non possano più essere modificate dopo la loro inizializzazione
- Si pensi ad esempio alla data di nascita di una persona, una volta istanziato un oggetto e inizializzata la sua data di nascita può non avere senso che tale data venga modificata
- Ogni oggetto di tipo persona avrà la sua data di nascita, parte del suo stato, il cui valore - diverso da quello di altri oggetti – **deve** essere inizializzato all'atto della creazione dell'oggetto

Variabili di istanza final



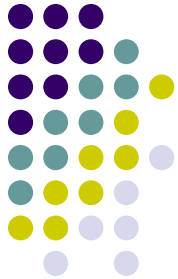
- Per ottenere questo effetto le variabili di istanza devono essere dichiarate final
- In questo modo una parte dello stato degli oggetti della classe sarà costante e allo stesso tempo il suo valore potrà essere diverso per ciascun oggetto, inizializzando la costante nei costruttori
- Nel caso invece si voglia che tutti gli oggetti abbiano lo stesso valore per la costante, questa può essere inizializzata all'atto della sua dichiarazione nella classe

Classe Counter: versione aggiornata

- Nella classe Counter in effetti finora abbiamo considerato modificabile il valore massimo che il contatore può raggiungere.
- Un utente può addirittura modificare questo valore con un metodo set. Questo può non avere senso nella realtà, ed anzi minare la consistenza degli oggetti.
  - Che succede se l'utente imposta la soglia ad un valore minore del valore già raggiunto dal contatore?
- Potrebbe essere più sensato che il valore massimo raggiungibile sia una costante, diversa per ogni contatore a seconda delle esigenze, ma costante.
- Questo richiede delle modifiche alla classe compresa l'eliminazione e l'uso della funzione set
- La prossima versione è aggiornata:
  - con le modifiche necessarie per rendere costante la variabile di istanza max
  - con l'aggiunta della variabile di classe (static) per il conteggio degli oggetti e la funzione statica per l'accesso a tale variabile.

Classe Counter – versione 5

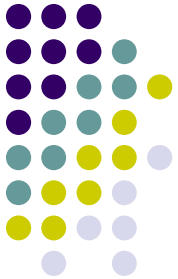
```
package runningexamples;

public class Counter {

    private long value;
    private final long max;
    private static long objcount;

    //costruttore senza argomenti
    public Counter() {max=10; value=0; objcount++;}
    //costruttore con argomenti
    public Counter(long m) {max=m; value=0; objcount++;}
    //costruttore di copia
    public Counter(Counter C) {max=C.max; value=C.value; objcount++;}
    // getter
    public long get_value() {return value;}
    public long get_max() {return max;}
    //metodi static
    public static long get_objcount() {return objcount;}
    // metodi per la gestione del contatore
    public void increment() {value++;}
    public void decrement() {value--;}
    //predicati
    public boolean ovf() {return (value+1)>=max;}
    public boolean unf() {return (value-1)<0;}
    public boolean equal(Counter C) {
          return ((max==C.max) && (value==C.value)); }
    public boolean not_equal(Counter C) {
          return !(equal(C));  }
}
```



Classe di prova – versione 5 (continua)

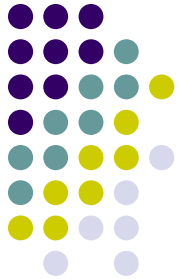
```
package runningexamples;

public class RunningExamples {

    public static void main(String[] args) {

        System.out.print("numero di oggetti contatore creati: " + Counter.get_objcount() + "\n");

        Counter b = new Counter();
        Counter c = new Counter(4);
        Counter d = new Counter(c);

        System.out.print("valore corrente del contatore b: " + b.get_value() + "\n");
        System.out.print("valore corrente del contatore c: " + c.get_value() + "\n");
        System.out.print("valore corrente del contatore d: " + d.get_value() + "\n");
        System.out.print("soglia corrente del contatore b: " + b.get_max() + "\n");
        System.out.print("soglia corrente del contatore c: " + c.get_max() + "\n");
        System.out.print("soglia corrente del contatore d: " + d.get_max() + "\n");

        if(!c.ovf()) {
            c.increment();
            System.out.print("valore corrente del contatore c dopo l'incremento: " + c.get_value() + "\n"); }
        else System.out.print("pre-condizione non verificata, impossibile incrementare c! \n");
        if(!c.unf()){
            c.decrement();
            System.out.print("valore corrente del contatore c dopo il decremento: " + c.get_value() + "\n"); }
            else System.out.print("pre-condizione non verificata, impossibile decrementare c! \n");
        if(!c.unf()){
            c.decrement();
            System.out.print("valore corrente del contatore c dopo il decremento: " + c.get_value() + "\n"); }
        else System.out.print("pre-condizione non verificata, impossibile decrementare c! \n");
```

Classe di prova – versione 5



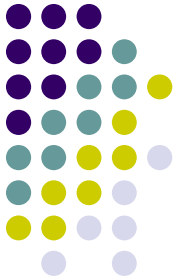
```
System.out.print("valore corrente della soglia massima di d: " + d.get_max() + "\n");

      // test del metodo equal
      System.out.print("valore corrente della soglia massima di c: " + c.get_max() + "\n");
      if(c.equal(d)) System.out.print("gli oggetti riferiti da c e d sono uguali \n");
      else System.out.print("gli oggetti riferiti da c e d sono diversi \n");
      if(c==d) System.out.print("c e d sono uguali, quindi puntano allo stesso oggetto \n");
      else System.out.print("c e d sono diversi, quindi puntano a oggetti diversi \n");
      System.out.print("numero di oggetti contatore creati: " + b.get_objcount() + "\n");

   }
}
```

Classi e Oggetti Immutabili



- Una classe che non fornisce funzioni che consentono all'utente di modificare lo stato di un oggetto è una classe immutabile, e le sue istanze sono dette oggetti immutabili
- Dopo essere stato creato un oggetto immutabile non è più modificabile dall'utente
- Una classe immutabile offre dei vantaggi, perché è possibile condividere riferimenti oggetti della classe senza mettere a rischio l'incapsulamento
- In particolare è pericoloso un metodo di accesso (get) che restituisce un riferimento a un oggetto che non sia di una classe immutabile.

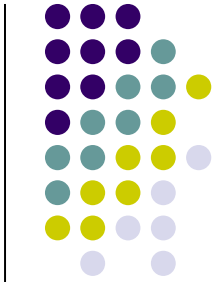
Esempio (1/2)



- Supponiamo di voler sviluppare la classe Persona che tra le sue variabili di istanza ha un riferimento ad un oggetto di tipo Data

```
class Persona
{
//variabili di istanza
private String nome;
private Data Birthday;

//metodi
…
public Data getBirthday(){return Birthday;}
…
}
```

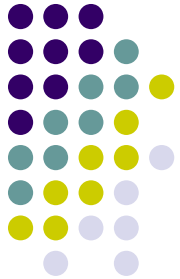
Esempio (2/2)



- Se la classe Data non è immutabile e fornisce ad esempio i metodi set che consentono di modificare giorno, mese ed anno, un utente della classe Persona **potrebbe modificare la variabile di istanza privata Birthday** utilizzando tali metodi sul riferimento ritornato da getBirthday!

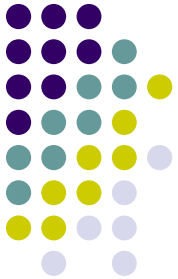
```
Persona mario = ...;
Data d = mario.getBirthday();
d.setGiorno(10); //cambia la data di nascita di mario!
```

Classi e Oggetti Immutabili



- In questi casi sarebbe opportuno che il metodo get non restituisse il riferimento all'oggetto, ma ad una copia dell'oggetto, cioè ad un suo clone (vedremo più avanti)
- In alternativa un modo per rendere immutabili delle variabili di istanza è dichiararle final (introducendo però evidenti limitazioni del loro uso anche da parte della classe)
- Come ultima osservazione notiamo che la variabile di istanza nome nell'esempio persona non presenta questo problema perché in Java le stringhe sono immutabili...

Riferimenti



- Programmare in Java: Capitolo 8, §8.3, §8.4, §8.5, §8.6, §8.7, §8.11, §8.13