

# Metodologie di Programmazione (M-Z)

Il semestre - a.a. 2022 – 2023

Parte 2 – Classi e Oggetti\*\*

a cura di Stefano Faralli\*



SAPIENZA  
UNIVERSITÀ DI ROMA

\*Tutti i diritti relativi al presente materiale didattico ed al suo contenuto sono riservati a Sapienza e ai suoi autori (o docenti che lo hanno prodotto). È consentito l'uso personale dello stesso da parte dello studente a fini di studio. Ne è vietata nel modo più assoluto la diffusione, duplicazione, cessione, trasmissione, distribuzione a terzi o al pubblico pena le sanzioni applicabili per legge.

\*\* I crediti sulle slide di questo corso sono riportati nell'ultima slide

# **Classi e Oggetti**

# Classi e oggetti

Possono:

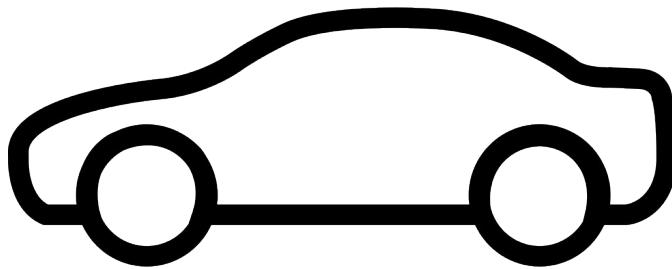
- Modellare gli oggetti del **mondo reale**
- Rappresentare oggetti **grafici**
- Rappresentare **entità software** (file, immagini, eventi, ecc.)
- Rappresentare **concetti astratti** (regole di un gioco, posizione di un giocatore)
- Rappresentare **stati** di un processo, di esecuzione, ecc.

# Classi e oggetti

- Una **classe** è un pezzo del codice sorgente di un programma che **descrive** un particolare tipo di oggetti
- Le classi vengono definite dal programmatore (**definizione di classe**)
- La **classe** fornisce un **prototipo astratto** per gli oggetti di un particolare tipo
- Ne definisce la struttura in termini di:
  - **Campi** (stato) degli oggetti
  - **Metodi** (comportamenti) degli oggetti
- Un **oggetto** è un'**istanza** (un esemplare) di una **classe**
- Un **programma** può creare e usare **uno o più oggetti** (istanze) della stessa classe

# Classe e Oggetto a confronto

Classe: automobile



• Campi:

- String `modello`;
- Color `colore`;
- int `numPasseggeri`;
- double `benzina`;

• Metodi:

- Aggiungi/togli passeggero
- Riempি serbatoio
- Segnala quantità benzina

Definizione/Progettazione

Oggetto: una certa automobile



• Campi:

- `String modello = "5";`
- `Color colore = Color.PINK;`
- `int numPasseggeri = 1;`
- `double benzina = 50;`

• Metodi:

Come la classe

Tangibile/Implementazione

# Classe e Oggetto a confronto

## Classe:

- Definita mediante parte del codice sorgente del programma
- Scritta dal programmatore

## Oggetto:

- Un'entità all'interno di un programma in esecuzione  
a runtime
- Creato quando un programma “gira” (dal metodo **main** o da un altro metodo)

# Classe e Oggetto a confronto

## Classe:

- Specifica la struttura (ovvero numero e tipi) dei campi dei suoi oggetti
- Specifica il comportamento dei suoi oggetti mediante il codice dei metodi

## Oggetto:

- Contiene specifici valori dei campi; i valori possono cambiare durante l'esecuzione  
a runtime
- Si comporta nel modo prescritto dalla classe quando il metodo corrispondente viene chiamato a tempo di esecuzione  
a runtime

# Classe e Oggetto a confronto



# Classe e Oggetto a confronto



# Classe e Oggetto a confronto

## Classe: Anello



### • Campi:

- int x;
- int y;
- int r;

### • Metodi:

- ruota

Definizione/Progettazione

Oggetti:

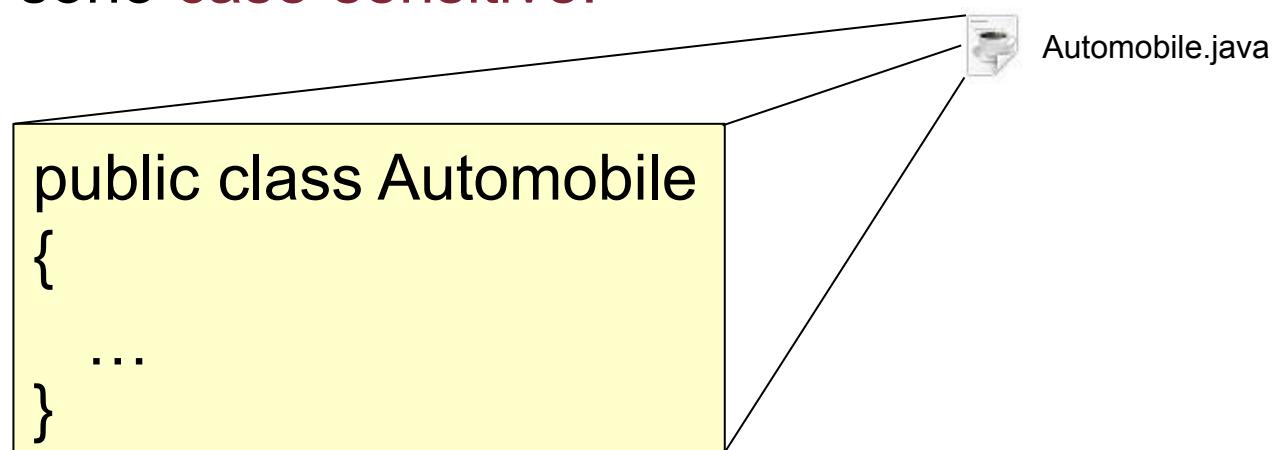
anello<sub>1</sub>, ..., anello<sub>182</sub>



a runtime

# Classi e file sorgenti

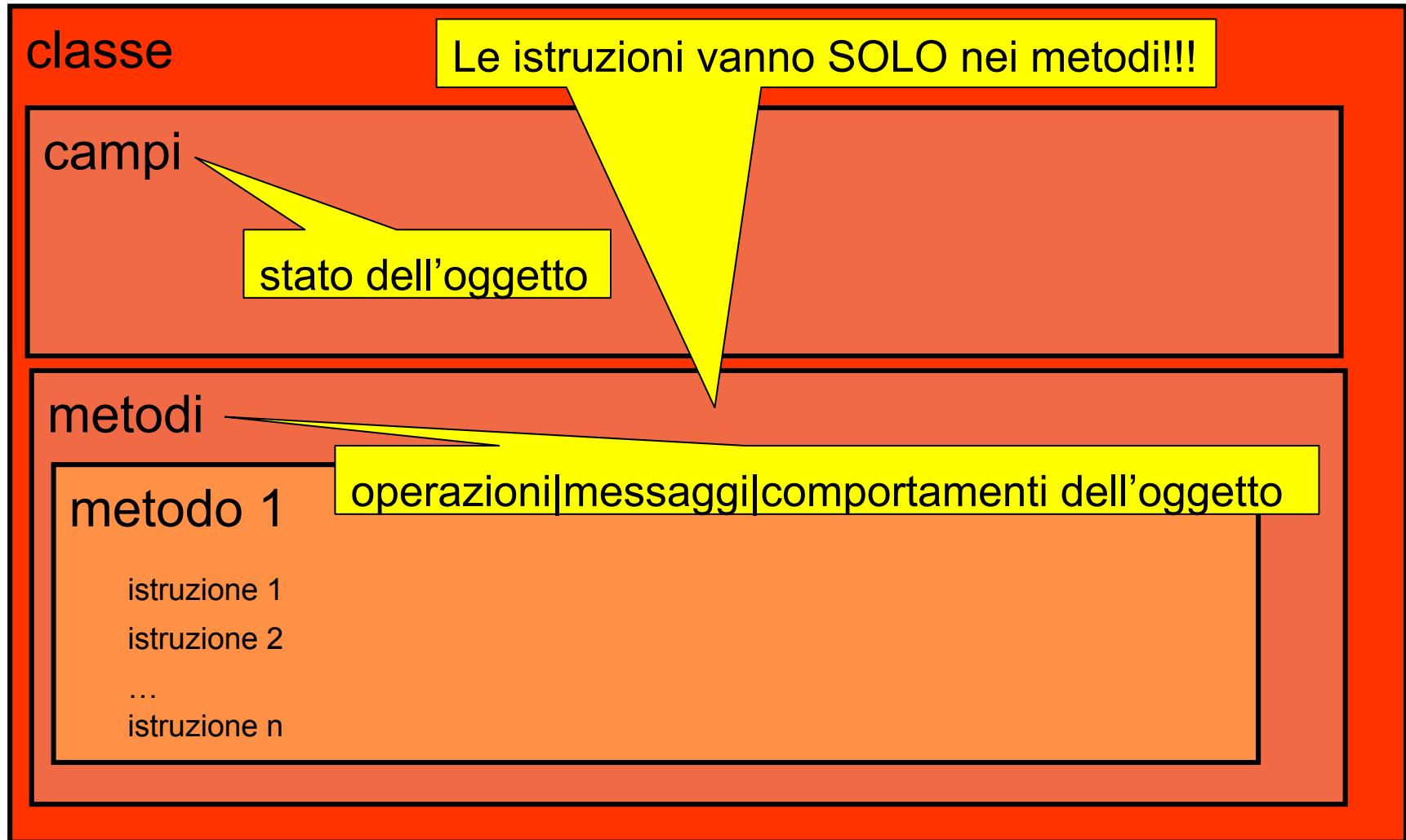
- Ogni **classe** è memorizzata in un **file** separato
- Il **nome del file** DEVE essere lo **stesso** della **classe**, con estensione **.java**
- I nomi di classe iniziano sempre con una maiuscola (es. **Automobile**, non **automobile**)
- I nomi in Java sono **case-sensitive!**



# Librerie

- I programmi Java normalmente non sono scritti **da zero**
- Esistono **migliaia di classi di libreria** per ogni esigenza (sia standard scritte da Sun/Oracle, sia sul Web scritte da centinaia di sviluppatori)
- Le **classi** sono organizzate in **package**
- Alcuni esempi:
  - **java.util** – classi di utilità
  - **java.awt** – classi per la grafica e le finestre
  - **javax.swing** e **javafx** – sviluppo di interfacce GUI
- Un package “**speciale**” è **java.lang**: contiene le classi fondamentali per la programmazione in Java (es. **String**, **System**, ecc.)

# Come strutturare il codice di una classe in Java?



## Esercizio: Un contatore

- Vogliamo realizzare una classe che rappresenta un contatore
- Il contatore permette di:
  - Incrementare il conteggio attuale
  - Otttenere il conteggio attuale
  - Resettare il conteggio a 0 (o a un altro valore)

# Counter.java

```
public class Counter
{
    /**
     * Valore intero del contatore
     */
    private int value; // Commento Javadoc  
Dichiarazione di un campo  
Tipicamente i campi sono privati

    /**
     * Costruttore della classe
     */
    public Counter() // Costruttore degli oggetti della classe  
Inizializza il campo value
    {
        value = 0;
    }

    /**
     * Incrementa il contatore
     */
    public void count() // I metodi della classe sono pubblici  
Incrementa il valore di value
    {
        value++;
    }

    /**
     * Ottiene il valore corrente del contatore
     * @return valore intero del contatore
     */
    public int getValue() { return value; } // Restituisce il valore di value
}
```

# Campi

- Un **campo** (detto anche **variabile di istanza**) costituisce la **memoria privata** di un oggetto
- Ogni **campo** ha un **tipo di dati** (es. il valore del contatore è intero)
- Ogni **campo** ha un **nome** fornito dal programmatore (es. `value` nella diapositiva precedente)

# Dichiarare un campo

- La dichiarazione di un campo avviene come segue:

```
private [static] [final] tipo_di_dati nome;
```

Tipicamente ad **accesso privato**

Se specificato indica che il  
campo è **condiviso da tutti**

Se specificato indica che il  
campo è **una costante**

Es. int, double, String, ecc.

# Esempi di dichiarazione

```
public class Hotel
{
    /**
     * Da evitare l'uso di una variabile "di comodo" come campo di una classe
     */
    private int k;

    /**
     * Nome dell'hotel
     */
    private String nome;

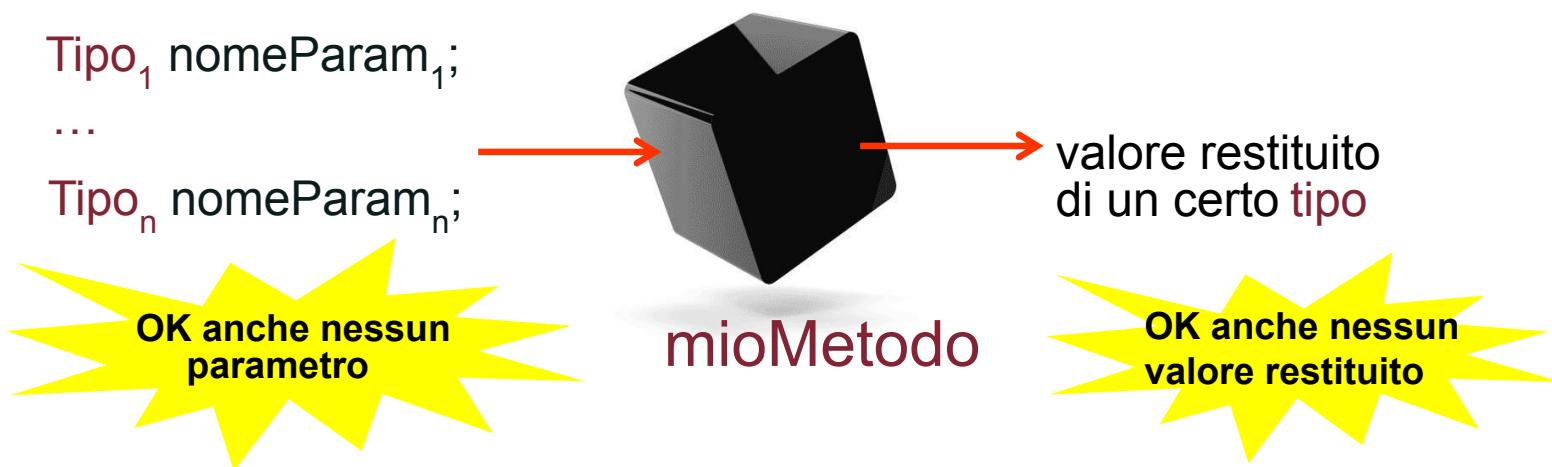
    /**
     * Numero di stanze
     */
    private int numeroStanze;

    /**
     * Superficie totale
     */
    private double superficie;

    /**
     * Sulla Guida Michelin
     */
    private boolean bGuidaMichelin;
}
```

# Metodi

- Un **metodo** è tipicamente **pubblico**, ovvero visibile a tutti
- Il **nome di un metodo** per convenzione inizia con una lettera minuscola, mentre le parole seguenti iniziano con lettera maiuscola (es. dimmiTuttoQuelloCheSai())
  - convenzione detta **CamelCase**



# Definizione di un metodo

- La definizione di un metodo avviene come segue:

```
public tipo_di_dati nomeDelMetodo(tipo_di_dati nomeParam1, ..., tipo_di_dati nomeParamn)  
{  
    istruzione 1;  
    .  
    .  
    .  
    istruzione m;  
}
```

Tipicamente ad **accesso pubblico**

Nome del metodo

Tipo del valore restituito (void se non viene restituito nulla)

Corpo del metodo (racchiuso da parentesi graffe)

Elenco (eventualmente vuoto) dei nomi di parametri con relativo tipo

# Definizione di un metodo

- La definizione di un metodo avviene come segue:

```
public tipo_di_dati nomeDelMetodo(tipo_di_dati nomeParam1, ..., tipo_di_dati nomeParamn)  
{  
    istruzione 1;  
    .  
    .  
    .  
    istruzione m;  
}
```

Tipicamente ad **accesso pubblico**

Nome del metodo

Tipo del valore restituito (void se non viene restituito nulla)

Corpo del metodo (racchiuso da parentesi graffe)

Elenco (eventualmente vuoto) dei nomi di parametri con relativo tipo

- Esempi:

```
public int getValue() { return value; }
```

```
public void reset(int newValue) { value = newValue; }
```

## Metodi e valori restituiti

- Un metodo può **restituire** un valore al chiamante

```
public int getValue() { return value; }
```

- La parola chiave **void** nell'**intestazione** (“signature” o “header”) del metodo indica che il metodo non restituisce alcun valore:

```
public void reset(int newValue) { value = newValue; }
```

# Costruttori

- Metodi (funzioni) per la **creazione** degli oggetti di una classe
- Possiedono **sempre** lo **stesso nome** della classe
- Inizializzano i **campi** dell'oggetto
- Possono prendere **zero, uno o più parametri**
- **NON** hanno valori di uscita (**MA** non specificano void)
- Una classe può avere anche **più costruttori** che differiscono nel numero e nei tipi dei parametri

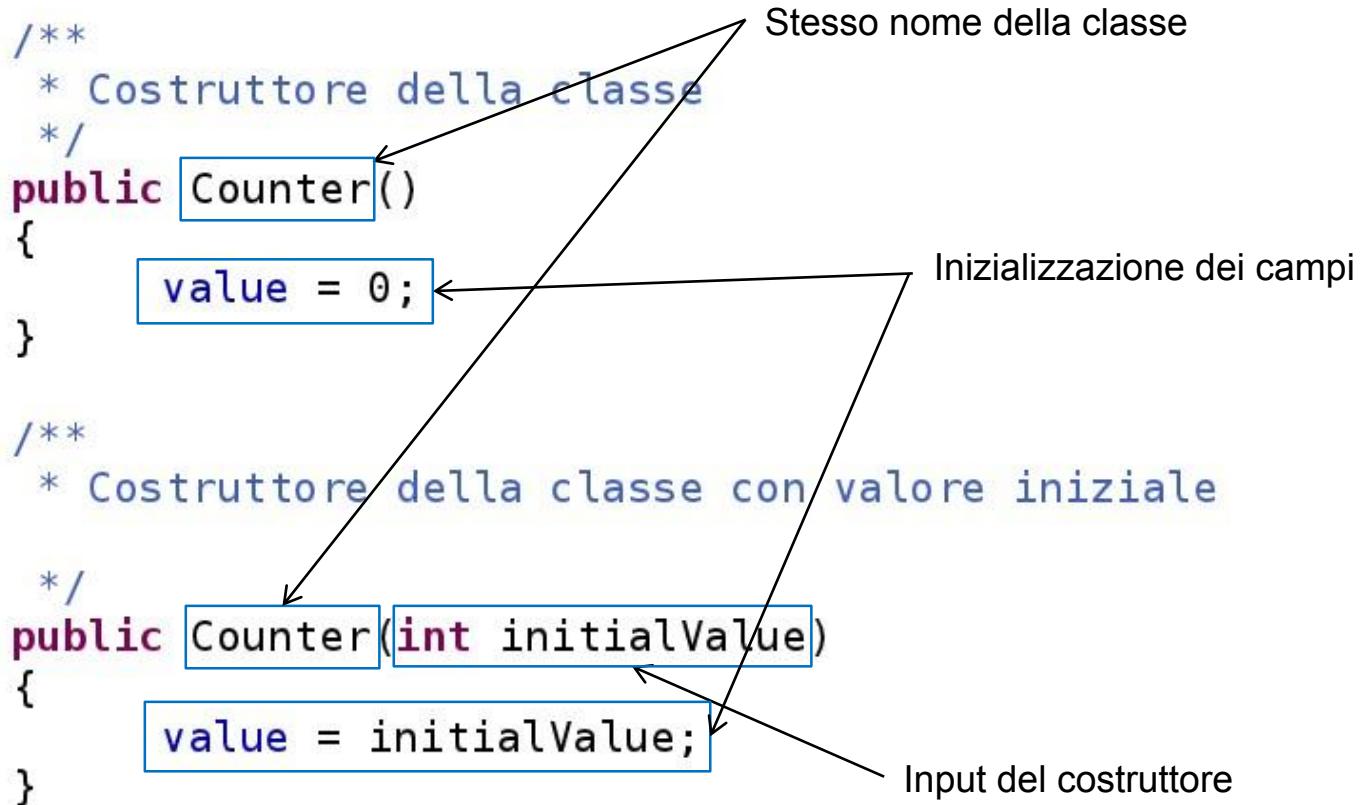
# Costruttori: esempio della classe Counter

```
/**  
 * Costruttore della classe  
 */  
public Counter()  
{  
    value = 0;  
}  
  
/**  
 * Costruttore della classe con valore iniziale  
 */  
public Counter(int initialValue)  
{  
    value = initialValue;  
}
```

Stesso nome della classe

Inizializzazione dei campi

Input del costruttore



# Costruttori: creazione dell'oggetto

- Un oggetto viene creato con l'operatore **new** :

```
static public void main(String[] args)
{
    Counter contatore1 = new Counter();
    Counter contatore2 = new Counter(42);

    System.out.println("Valore del contatore1: "+contatore1.getValue());
    System.out.println("Valore del contatore2: "+contatore2.getValue());
}

/**
 * Costruttore della classe con valore iniziale
 */
public Counter(int initialValue)
{
    value = initialValue;
}
```

operatore new

Il numero, l'ordine e i tipi dei parametri **devono corrispondere**

## Ancora sui costruttori

- Non è obbligatorio specificare un costruttore
- Se non ne viene specificato uno, Java crea per ogni classe un **costruttore di default** “vuoto” (senza parametri)
  - Inizializza le variabili d’istanza ai valori di default

# Implementiamo il metodo reset

- Versione 1: semplicemente azzera il contatore

```
public void reset() { value = 0; }
```

- Versione 2: reimposta il contatore a un determinato valore

```
public void reset(int newValue) { value = newValue; }
```



# Chiamate di Metodi

- I metodi vengono chiamati su un particolare oggetto:

```
static public void main(String[] args)
{
    Counter contatore1 = new Counter();
    Counter contatore2 = new Counter(42);

    contatore1.count();
    contatore2.count();

    contatore2.reset();
    contatore1.reset(10);

    System.out.println("Valore del contatore1: "+contatore1.getValue());
    System.out.println("Valore del contatore2: "+contatore2.getValue());
}
```

## Chiamate di Metodi

- Il numero e i tipi dei parametri (argomenti) passati a un metodo deve coincidere con i parametri formali del metodo

```
public void reset() { value = 0; }
public void reset(int newValue) { value = newValue; }
static public void main(String[] args)
{
    Counter contatore1 = new Counter();
    Counter contatore2 = new Counter(42);

    contatore1.count();
    contatore2.count();

    contatore2.reset();
    contatore1.reset(10);

    System.out.println("Valore del contatore1: "+contatore1.getValue());
    System.out.println("Valore del contatore2: "+contatore2.getValue());
}
```

# Variabili locali vs. campi

- I **campi** sono variabili dell'oggetto
  - Sono visibili **almeno** all'interno di tutti gli oggetti della stessa classe
  - Esistono per tutta la vita di un oggetto
- Le **variabili locali** sono variabili definite all'interno di un metodo
  - Come parametri del metodo o all'interno del corpo del metodo
  - Esistono dal momento in cui sono definite fino al termine dell'esecuzione della chiamata al metodo in questione

## Esercizio

- Progettare una classe **Persona** i cui oggetti rappresentano una persona e ne memorizzano il nome e il cognome
  - La classe espone un metodo **main** che crea un'istanza della Persona
  - La classe espone anche un metodo **stampa** che visualizza nome e cognome della persona

## Esercizio

- Progettare una classe **Quadrato**, i cui oggetti sono costruiti con il lato dello stesso
  - La classe è dotata di un metodo **getPerimetro** che restituisce il perimetro
  - E di un metodo **main** che crea un quadrato di lato 4 e ne stampa a video il perimetro

## Esercizio

- Progettare una classe **Cerchio** i cui oggetti rappresentano un cerchio
  - La classe è dotata dei metodi **getCirconferenza** e **getArea** (si usi la costante **Math.PI**)
  - La classe espone anche un metodo **main** che crea due cerchi (di raggio 1 e di raggio 5) e ne stampa la circonferenza (per il primo) e l'area (per il secondo)

## Esercizio

- Progettare una classe **BarraDiCompletamento** i cui oggetti rappresentano una barra di caricamento
  - Gli oggetti vengono costruiti con la percentuale di partenza
  - La classe espone un metodo **incrementa** che, data una percentuale in input, incrementa la percentuale di partenza con quella fornita in input (ad es. **new BarraDiCompletamento(5).incrementa(10)** porta la barra al 15%)
  - Il metodo **toString** dell'oggetto restituisce una stringa contenente la percentuale di completamento arrotondata con **Math.round()**
  - Il metodo **main** che crea una barra di completamento che parte da 0, la incrementa prima di 20 punti percentuale e poi di altri 25 e quindi stampa la rappresentazione stringa della barra

## Esercizio

- Progettare una classe **Rettangolo** i cui oggetti rappresentano un rettangolo e sono costruiti a partire dalle coordinate x, y e dalla lunghezza e altezza del rettangolo
- La classe implementa i seguenti metodi:
  - **trasla** che, dati in input due valori x e y, trasla le coordinate del rettangolo dei valori orizzontali e verticali corrispondenti
  - **toString** che restituisce una stringa del tipo “(x1, y1)->(x2, y2)” con i punti degli angoli in alto a sinistra e in basso a destra del rettangolo
- Implementare una classe di test **TestRettangolo** che verifichi il funzionamento della classe **Rettangolo** sul rettangolo in posizione (0, 0) e di lunghezza 20 e altezza 10, traslandolo di 10 verso destra e 5 in basso

## Esercizio

Progettare una classe **Colore** i cui oggetti rappresentano un colore in modalità RGB e che sono costruiti a partire da tre valori: R (rosso), G (verde), B (blu), ognuno dei quali ammette un valore intero nell'intervallo 0-255.

- La classe Colore espone anche due costanti BIANCO e NERO
- Fare in modo che ogni Rettangolo (vedere esercizio precedente) abbia associato un Colore di base NERO e che sia possibile impostare il colore di un rettangolo mediante un apposito metodo

# **Incapsulamento**

# Incapsulamento

- Perché utilizziamo le parole chiave **public** e **private**?
- Per nascondere le informazioni (“**information hiding**”) all’utente
- Il processo che nasconde i dettagli realizzativi, rendendo pubblica un’interfaccia, prende il nome di incapsulamento
  - Dettagli realizzativi: campi e implementazione
  - Interfaccia pubblica: metodi pubblici

# Perché encapsulare?

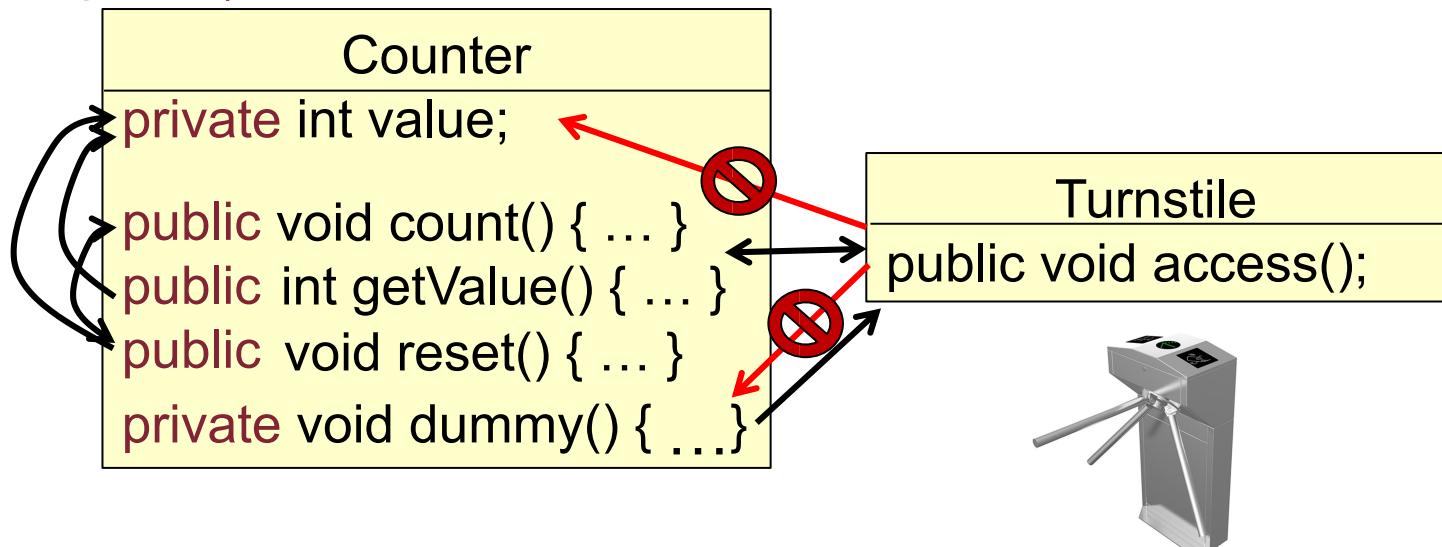
- Si semplifica e modularizza il lavoro di sviluppo assumendo un certo funzionamento a “scatola nera”
- Non è necessario sapere tutto, soprattutto molti inutili dettagli
- L'incapsulamento facilita il lavoro di gruppo e l'aggiornamento del codice (maintenance)
- Aiuta a rilevare errori: in presenza di moltissime classi, un certo errore si verifica solo in una determinata classe per cui ci si può concentrare su di essa

# Come interagiscono le classi tra loro?

- Una classe interagisce con le altre **principalmente** (**=quasi solo**) attraverso i **costruttori** e i **metodi pubblici**
- Le altre classi non devono conoscere i dettagli implementativi di una classe per usarla in modo efficace

## Accesso a campi e metodi (inclusi i costruttori)

- I campi e i metodi possono essere **pubblici**, **privati** (o **protetti**, come vedremo più in là)
- I metodi di una classe possono chiamare i metodi pubblici e privati della stessa classe
- I metodi di una classe possono chiamare i metodi pubblici (ma non quelli privati) di altre classi



# UML parte 1: i diagrammi delle classi



WIKIPEDIA  
L'enciclopedia libera

Voce Discussione

Leggi

Modifica

Modifica wikitesto

Cronologia

Cerca in Wikipedia



Accesso non effettuato discussioni contributi registrati entra

## Class diagram

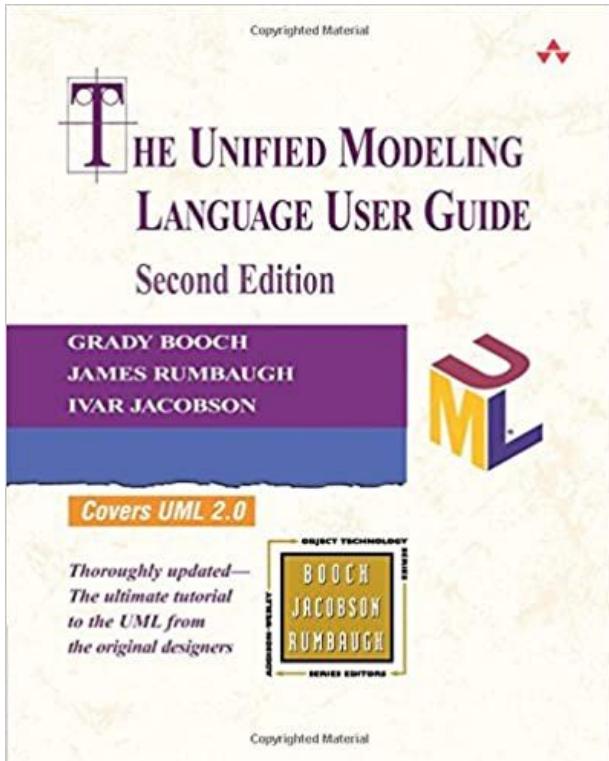
Da Wikipedia, l'enciclopedia libera.

I *diagrammi delle classi* (**class diagram**) sono uno dei tipi di *diagrammi* che possono comparire in un *modello UML*. In termini generali, consentono di descrivere *tipi di entità*, con le loro caratteristiche e le eventuali relazioni fra questi tipi. Gli strumenti concettuali utilizzati sono il concetto di *classe* del *paradigma object-oriented* e altri correlati (per esempio la *generalizzazione*, che è una relazione concettuale assimilabile al meccanismo *object-oriented* dell'*ereditarietà*).

In *ingegneria del software*, **UML** (*Unified Modeling Language*, "linguaggio di modellizzazione unificato") è un *linguaggio di modellazione e di specifica* basato sul *paradigma orientato agli oggetti*. Il nucleo del linguaggio fu definito nel 1996 da *Grady Booch*, *Jim Rumbaugh* e *Ivar Jacobson* (detti "i tre amigos") sotto l'egida dell'*Object Management Group* (OMG), consorzio che tuttora gestisce lo standard UML.



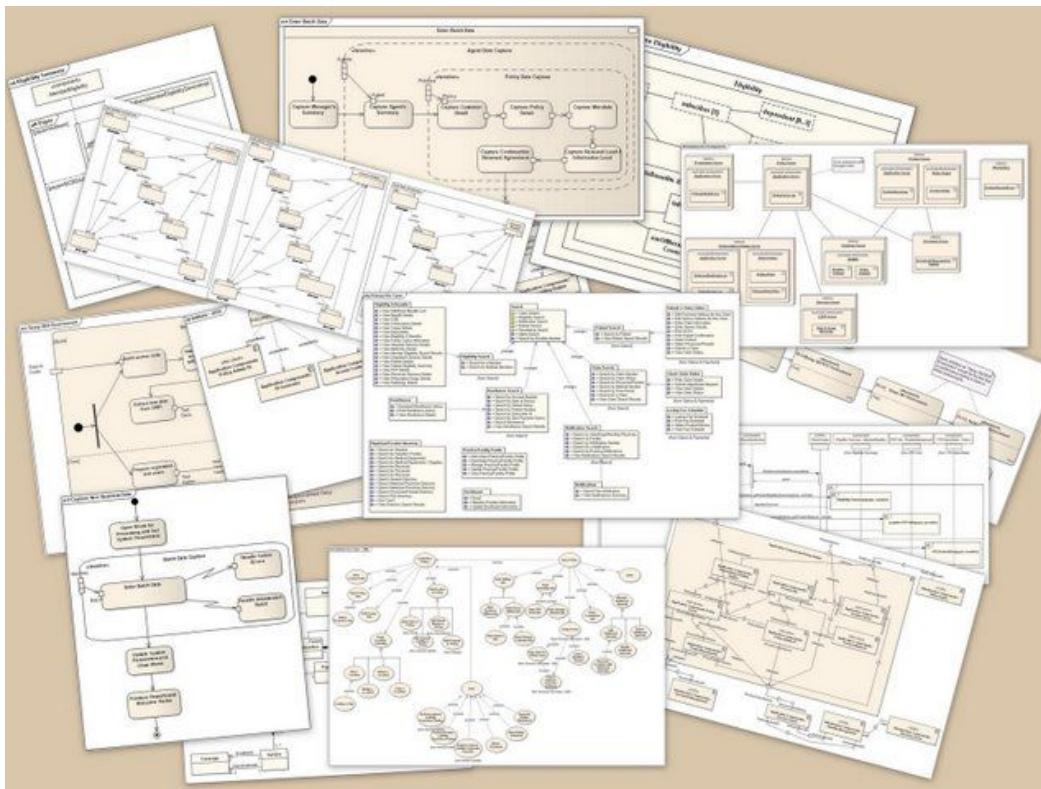
# UML parte 1: i diagrammi delle classi



[https://personal.utdallas.edu/~chung/Fujitsu/UML\\_2.0/Rumbaugh-UML\\_2.0\\_Reference\\_CD.pdf](https://personal.utdallas.edu/~chung/Fujitsu/UML_2.0/Rumbaugh-UML_2.0_Reference_CD.pdf)



# UML parte 1: i diagrammi delle classi



# UML parte 1: i diagrammi delle classi

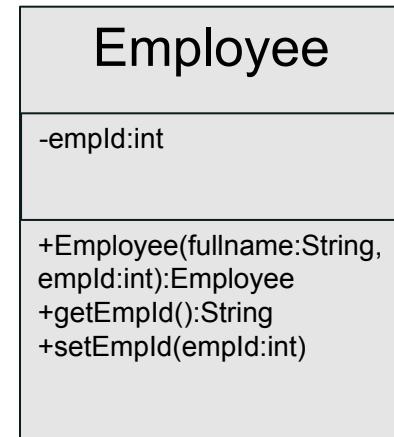
Object Oriented Analysis

Reverse Engineering

Entità  
**(Oggetti** nel mondo  
reale)



Classe in UML  
modello



Classe in  
Java

```
1 // Java bean for Employee
2 public class Employee extends Person {
3     // Private variable
4     private int empId;
5     // Constructor
6     public Employee(String fullName, int empId) {
7         super(fullName);
8         setEmpId(empId);
9     }
10    // Getter and setter for variable
11    public int getEmpId() {
12        return empId;
13    }
14    public void setEmpId (int empId) {
15        this.empId=empId;
16    }
17 }
```

Oggetti in Java

**new** Employee("John",...);  
**new** Employee("Dessy",...);

# UML parte 1: i diagrammi delle classi

Nome del Package 1



Nome del Package 2



In questo Diagramma abbiamo un package che include una classe e un sotto package



# UML parte 1: i diagrammi delle classi

```
23 public class Rettangolo {  
24     private double x, y, lunghezza;  
25     private double altezza;  
26  
27     public Rettangolo(double x, double y, double lunghezza, double altezza)  
28     {  
29         this.x=x;  
30         this.y=y;  
31         this.lunghezza=lunghezza;  
32         this.altezza=altezza;  
33     }  
34  
35     public void trasla(double x, double y)  
36     {  
37         this.x+=x;  
38         this.y+=y;  
39     }  
40  
41     public String toString()  
42     {  
43         double x2=x+lunghezza;  
44         double y2=y+altezza;  
45         return "("+x+", "+y+")-> (" +x2+", "+y2+")" ;  
46     }  
47 }
```

## Rettangolo

-x:double  
-y:double  
-lunghezza:double  
-altezza:double

- + Rettangolo(x:double,...):Rettangolo
- + trasla(x:double, y:double)
- + toString():String

# UML parte 1: i diagrammi delle classi

- + (visibilità pubblica): ogni elemento che può accedere alla classe può anche accedere a ogni suo membro con visibilità pubblica
- (visibilità privata): solo le operazioni della classe possono accedere ai membri con visibilità privata
- # (visibilità protetta): solo le operazioni appartenenti alla classe o ai suoi discendenti possono accedere ai membri con visibilità protetta
- ~ (visibilità package): ogni elemento nello stesso package della classe (o suo sottopackage annidato) può accedere ai membri della classe con visibilità package

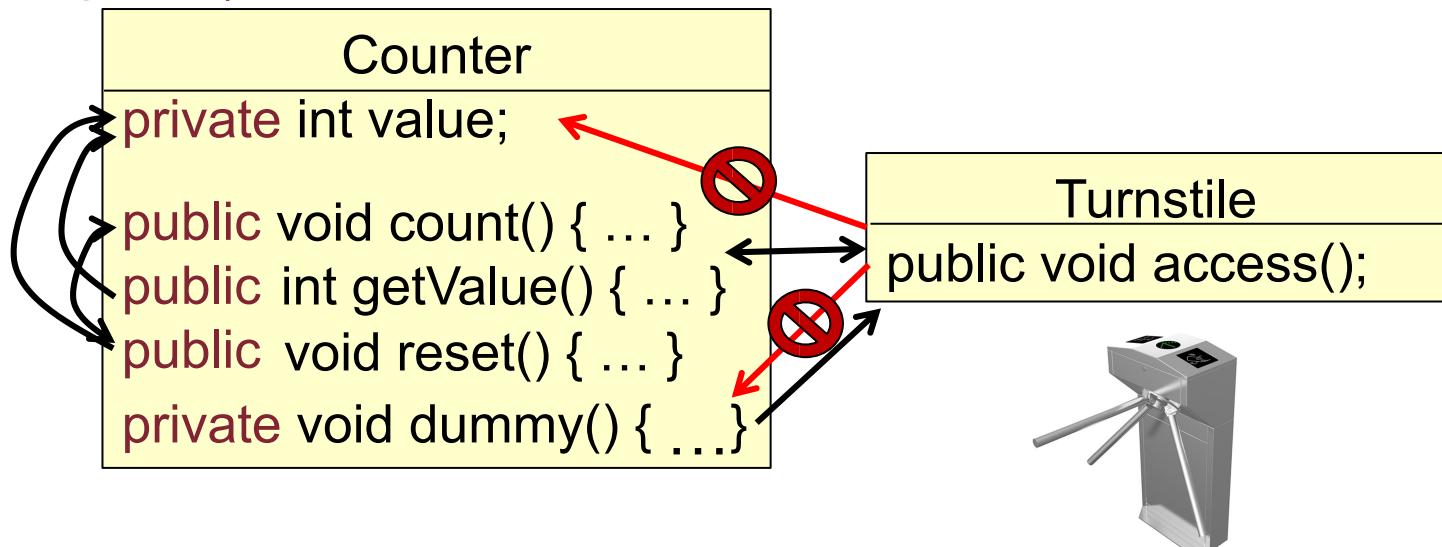
## Rettangolo

-x:double  
-y:double  
-lunghezza:double  
-altezza:double

- + Rettangolo(x:double,...):Rettangolo
- + trasla(x:double, y:double)
- + toString():String

## Accesso a campi e metodi (inclusi i costruttori)

- I campi e i metodi possono essere **pubblici**, **privati** (o **protetti**, come vedremo più in là)
- I metodi di una classe possono chiamare i metodi pubblici e privati della stessa classe
- I metodi di una classe possono chiamare i metodi pubblici (ma non quelli privati) di altre classi



## Un esempio: realizzare un semplice menù

- Dovete realizzare la classe **Menu**, in grado di visualizzare un menù come questo:

- 1) Inizia il gioco
- 2) Carica gioco
- 3) Aiuto
- 4) Esci

## Fase 1: identifica i metodi richiesti

- Aggiungere una nuova opzione
- Visualizzare il menù

## Fase 2: specifica l'interfaccia pubblica

- Aggiungere una nuova opzione:

```
public void addOption(String option) { }
```

- Visualizzare il menù:

```
public void display() { }
```

- Costruire l'oggetto:

- Costruttore con una prima opzione in input?
  - Costruttore vuoto?

- Meglio evitare casi speciali:

```
public Menu() { }
```

## Fase 3: scrivere la documentazione per l'interfaccia pubblica

```
/**  
 * Un menu' che viene visualizzato in una finestra di console  
 * @author navigli  
 *  
 */  
public class Menu  
{  
    /**  
     * Costruisce un menu' vuoto (senza opzioni)  
     */  
    public Menu()  
    {  
    }  
  
    /**  
     * Aggiunge un'opzione alla fine del menu'  
     * @param option l'opzione da aggiungere  
     */  
    public void addOption(String option)  
    {  
    }  
  
    /**  
     * Visualizza il menu' su console  
     */  
    public void display()  
    {  
    }  
}
```

## Fase 4: Identifica i campi

- In ogni momento, qual è lo stato di un oggetto di tipo **Menu**?

```
public class Menu
{
    private String menuText;
    private int optionCount;

    ...
}
```

## Fase 5: Implementa i metodi

```
public class Menu
{
    ...

    /**
     * Costruisce un menu' vuoto (senza opzioni)
     */
    public Menu()
    {
        menuText = "";
        optionCount = 0;
    }

    /**
     * Visualizza il menu' su console
     */
    public void display()
    {
        System.out.println(menuText);
    }

    /**
     * Aggiunge un'opzione alla fine del menu'
     * @param option l'opzione da aggiungere
     */
    public void addOption(String option)
    {
        optionCount++;
        menuText += optionCount + ") " + option + "\n";
    }
}
```

## Fase 6: Collauda la classe

- Scriviamo un'altra classe per “testare” (collaudare) la nostra:

```
public class MenuTest
{
    static public void main(String[] args)
    {
        Menu menu = new Menu();

        menu.addOption("Open new account");
        menu.addOption("Log into existing account");
        menu.addOption("Help");
        menu.addOption("Quit");

        menu.display();
    }
}
```

## Esercizio 1/2

- Progettare una classe **Programmatore** i cui oggetti rappresentano persone che svolgono il lavoro di sviluppo presso un'azienda
- La classe implementa i seguenti metodi:
  - un costruttore con il nome e cognome della persona
  - un metodo **setAzienda** che imposta il nome dell'azienda per cui la persona lavora
  - un metodo **addLinguaggio** che aggiunge un linguaggio di programmazione a quelli inizialmente usati dal programmatore
  - i metodi **getNome**, **getCognome** e **getAzienda** che restituiscono i valori corrispondenti
  - un metodo **getLinguaggi** che restituisce la stringa dei linguaggi (separati da spazio) noti al programmatore

## Esercizio 2/2

- Implementare un main all'interno della stessa classe in modo che il seguente codice funzioni correttamente:

```
Programmatore p1 = new Programmatore("Bjarne", "Stroustrup");
Programmatore p2 = new Programmatore("Brian", "Kernighan");
Programmatore p3 = new Programmatore("James", "Gosling");

p1.addLinguaggio("C");
p1.addLinguaggio("C+");
p1.setAzienda("Morgan Stanley");

p2.addLinguaggio("C");
p2.addLinguaggio("AWK");

p3.addLinguaggio("Java");
p3.setAzienda("Oracle");

// stampa: Morgan Stanley
System.out.println(p1.getAzienda());
// stampa: C AWK
System.out.println(p2.getLinguaggi());
```

# **java.lang.String**

# La classe `java.lang.String`

- Una **classe fondamentale**, perché è relativo a un tipo di dato i cui letterali sono parte della sintassi del linguaggio e per il quale il **significato dell'operatore + è ridefinito**
- Non richiede **import** perché appartiene al package “speciale” `java.lang`

# Ottenere la lunghezza di una stringa

- La classe String è dotata del metodo **length()**

Ad esempio:

```
String s = "ciao";
```

```
System.out.println(s.length());
```

- Stamperà il valore 4, che è pari al numero di caratteri di cui è costituita la stringa s

## Stringa tutta in maiuscolo o tutta in minuscolo

- Con i metodi `toLowerCase()` e `toUpperCase()` si ottiene un'**altra** stringa tutta **minuscola** o **maiuscola**, rispettivamente
- La stringa su cui viene invocato il metodo **non viene modificata**
- Ad esempio:

```
String min = "Ciao".toLowerCase(); // "ciao"
```

```
String max = "Ciao".toUpperCase(); // "CIAO"
```

```
String ariMin = max.toLowerCase(); // "ciao"
```

# Ottenere singoli caratteri

E' possibile ottenere il k-esimo carattere di una stringa con il metodo **charAt**

- **Importante:**
  - il **primo carattere** è in posizione **0**
  - l'**ultimo carattere** è in posizione **stringa.length()-1**
- Esempio:

```
String s = "ciao";
```

```
System.out.println(s.charAt(2));
```

stamperà il carattere 'a' (si noti che charAt restituisce un carattere (tipo char), non una stringa)

# Ottenere una sottostringa

- E' possibile ottenere una sottostringa di una stringa con il metodo **substring(startIndex, endIndex)**, dove:
  - **startIndex** è l'indice di partenza della sottostringa
  - **endIndex** è l'indice successivo all'ultimo carattere della sottostringa
- Ad esempio:

```
String s = "ciao";
```

```
System.out.println(s.substring(1, 3));
```

- Stamperà la stringa “ia”, dalla posizione 1 ('i') alla posizione 3 ('o') **esclusa**

Esiste anche una versione **substring(startIndex)** equivalente a **substring(startIndex, stringa.length())**

## Concatenare stringhe

- La concatenazione tra due stringhe può essere effettuata con l'operatore “speciale” **+** oppure mediante il metodo **concat(s)**

```
String s3 = s1+s2;
```

```
String s4 = s1.concat(s2);
```

- Tuttavia, se si devono concatenare parecchie stringhe, è bene utilizzare la classe **StringBuilder**, dotata dei metodi **append(String s)** e **insert(int posizione, String s)**

```
StringBuilder sb = new StringBuilder();
```

```
sb.append(s1).append(s2);
```

```
String s5 = sb.toString();
```

## Cercare in una stringa

- Si può cercare la (prima) **posizione** di un carattere **c** con **indexOf(c)**
  - restituisce -1 se il carattere non è presente
- E' possibile anche cercare la prima posizione di una sottostringa con **indexOf(s)** dove s è di tipo String
- Ad esempio:

```
int k = "happy happy birthday".indexOf('a');
```

```
int j = "din din don don".indexOf("don");
```

```
int h = "abcd".indexOf('e');
```

- k varrà 1, j varrà 8, mentre h varrà -1
- Con i metodi **startsWith** e **endsWith** è possibile verificare prefissi o suffissi della stringa

## Sostituire caratteri e sottostringhe

- Con il metodo **replace** è possibile sostituire tutte le occorrenze di un carattere o di una stringa all'interno di una stringa
- Esempio:

```
// s1 vale “uno due tre”
```

```
String s1 = “uno_due_tre”.replace(‘_’, ‘ ’);
```

```
// s2 vale “uno two tre”
```

```
String s2 = “uno due tre”.replace(“due”, “two”);
```

```
// s3 vale “aano daae tre”
```

```
String s3 = “uno due tre”.replace(“u”, “aa”);
```

# Confrontare stringhe

- Le stringhe, come peraltro tutti gli altri oggetti, vanno **SEMPRE** confrontate con il metodo **equals**
- Che differenza c'è tra **equals** e **==**?
  - L'operatore **==** confronta il riferimento (diciamo, l'indirizzo in memoria), quindi è **true se e solo se** si confrontano gli stessi oggetti fisici
  - L'operatore **equals** confronta la stringa carattere per carattere e restituisce **true** se le stringhe contengono la stessa sequenza di caratteri
- Ad esempio:

```
String s1 = "ciao", s2 = "ci"+ "ao", s3 = "hello";
```

```
System.out.println(s1 == s2); // potrebbe restituire false
```

```
System.out.println(s1.equals(s2)); // restituisce true
```

```
System.out.println(s1.equals(s3)); // restituisce false
```

# Spezzare le stringhe

- Il metodo **split** prende in input un'**espressione regolare s** (senza entrare in dettagli, è sufficiente pensarla come una semplice stringa) e restituisce un array di sottostringhe separate da s
- Esempio:

```
String[] parole = "uno due tre".split(" ");
```

```
// parole contiene l'array new String[] { "uno", "due", "tre" }
```

# Quindi se volessi aggiungere un metodo Menu.getOption()?

- Aggiungere un metodo `getOption` alla classe `Menu` che, dato un intero `k`, restituisca la `k`-esima opzione

```
/**  
 * Restituisce la k-esima opzione del menù  
 * @param k la posizione dell'opzione  
 * @return la stringa associata all'opzione specificata  
public String getOption(int k)  
{  
    String[] opzioni = menuText.split("\n");  
    String opzione = opzioni[k-1];  
    int posizione = opzione.indexOf(" ");  
    return opzione.substring(posizione+2);  
}
```

## Esempio: la Stringa “magica”

- Progettare una classe **Stringa42**, costruita a partire da 3 stringhe in input, che concatensi le stringhe inframezzate da spazi e conservi solo i primi 42 caratteri della stringa risultante
- La classe deve poter:
  - restituire la stringa di lunghezza massima 42
  - restituire l'iniziale di tale stringa
  - restituire un booleano che indichi se la stringa è pari alla terna di numeri “magici” 42 42 42
  - restituire un booleano che indichi se la stringa contiene il numero “magico” 42

# Esempio: la Stringa “magica”

```
public class Stringa42
{
    private String stringa;

    public Stringa42(String s1, String s2, String s3)
    {
        // equivalente a s1+" "+s2+" "+s3
        stringa = s1.concat(" ").concat(s2).concat(" ").concat(s3);

        // massima lunghezza 42
        if (stringa.length() > 42) stringa = stringa.substring(0, 42);
    }

    public String getStringa()
    {
        return stringa;
    }

    public char getIniziale()
    {
        return stringa.charAt(0);
    }

    public boolean isMagic()
    {
        return stringa.equals("42");
    }

    public boolean containsMagic()
    {
        return stringa.indexOf("42") != -1;
    }

    public static void main(String[] args)
    {
        Stringa42 s = new Stringa42("La risposta", "è'", "42");
        System.out.println(s.getIniziale());
        System.out.println(s.getStringa());
        System.out.println(s.isMagic());
        System.out.println(s.containsMagic());
    }
}
```

## Esercizio: registratore di cassa

- Progettare una classe che costituisca un modello di registratore di cassa
- La classe deve consentire a un cassiere di inserire i prezzi degli articoli e, data la quantità di denaro pagata dal cliente, calcolare il resto dovuto
- Quale stato (campi) dovete prevedere?
- Quali metodi vi servono?
  - Registra il prezzo di vendita per un articolo
  - Conclude la transazione, calcola il resto dovuto al cliente nel momento in cui effettua il pagamento e restituendolo in uscita

## Esercizio: punti e segmenti

- Progettare una classe **Punto** per la rappresentazione di un punto nello spazio tridimensionale
- E una classe **Segmento** per rappresentare un segmento nello spazio tridimensionale
- Scrivere una **classe di test** che crei:
  - due oggetti della classe Punto con coordinate (1, 3, 8) e (4, 4, 7)
  - un oggetto della classe Segmento che rappresenti il segmento che unisce i due punti di cui sopra

# Tipi di dato in Java: valori primitivi vs. oggetti

- E' importante tenere a mente la differenza tra:
  - Valori di tipo **primitivo** (int, char, boolean, float, double, ecc.)
  - Oggetti (istanze delle classi)
- La loro rappresentazione in memoria è differente:
  - **Valori primitivi**: memoria allocata automaticamente a tempo di compilazione
  - **Oggetti**: memoria allocata durante l'esecuzione del programma (operatore **new**)

## Inizializzazioni implicite per i campi della classe

- Al momento della creazione dell'oggetto i campi di una classe sono inizializzati automaticamente

Tipo del campo	Inizializzato implicitamente a
int, long	0, 0L
float, double	0.0f, 0.0
char	'\0'
boolean	false
classe X	null



ricordate NULL in C?

- ATTENZIONE: le inizializzazioni sono automatiche per i campi di una classe, ma **NON** per le variabili locali dei metodi

## Ancora sulle notazioni dei letterali

- Potete dichiarare un intero in notazione decimale:

```
int val = 42;
```

- Un intero in esadecimale:

```
int val = 0x2A;
```

- Un intero in binario: `int val = 0b101010;`
- Il valore di quei 4 byte **sarà sempre lo stesso**: 42

## Ancora sulle notazioni dei letterali

- Potete dichiarare un double in notazione decimale:

```
double val = 42.5;
```

- Un double in notazione scientifica:

```
double val = 0.425e2;
```

- Il float equivalente: **float** val = 42.5f;
- Il valore di quei 8 byte (o 4 byte, se float) **sarà sempre lo stesso**: 42.5

# Esempio di inizializzazione implicita di campi

```
public class Room
{
    /**
     * Contiene il numero di persone attualmente nella stanza
     */
    private int numberOfPeople;

    /**
     * Conta il numero di accessi alla stanza
     */
    private Counter accessCounter;

    public static void main(String[] args)
    {
        int n;
        Counter myCounter;

        System.out.println(n+" "+myCounter);

        Room room = new Room();
        System.out.println(room.numberOfPeople);
        System.out.println(room.accessCounter);
    }
}
```

Crea un nuovo oggetto in memoria contenente i campi *numberOfPeople* e *accessCounter* **inizializzati**

non inizializzato esplicitamente  
idem...

Errore a tempo di compilazione  
**variable *n* may not have been initialized**

Stampa 0

Stampa null

# Riferimenti e oggetti

- Per chi sa cosa siano i puntatori, i riferimenti sono ciò che ne è rimasto in Java
- Un riferimento è un indirizzo di memoria
  - Tuttavia non conosciamo il valore numerico dell'indirizzo
- Quindi gli oggetti non sono mai memorizzati direttamente nelle variabili, ma solo mediante il loro riferimento

# Variabili di tipi primitivi e variabili “riferimento”

- Le variabili contengono:
  - valori di tipi di dati primitivi
  - oppure riferimenti a oggetti
- Non esistono variabili che contengono oggetti
  - Analogamente ai puntatori in C

# Oggetti: i tre passi della dichiarazione, creazione e assegnazione

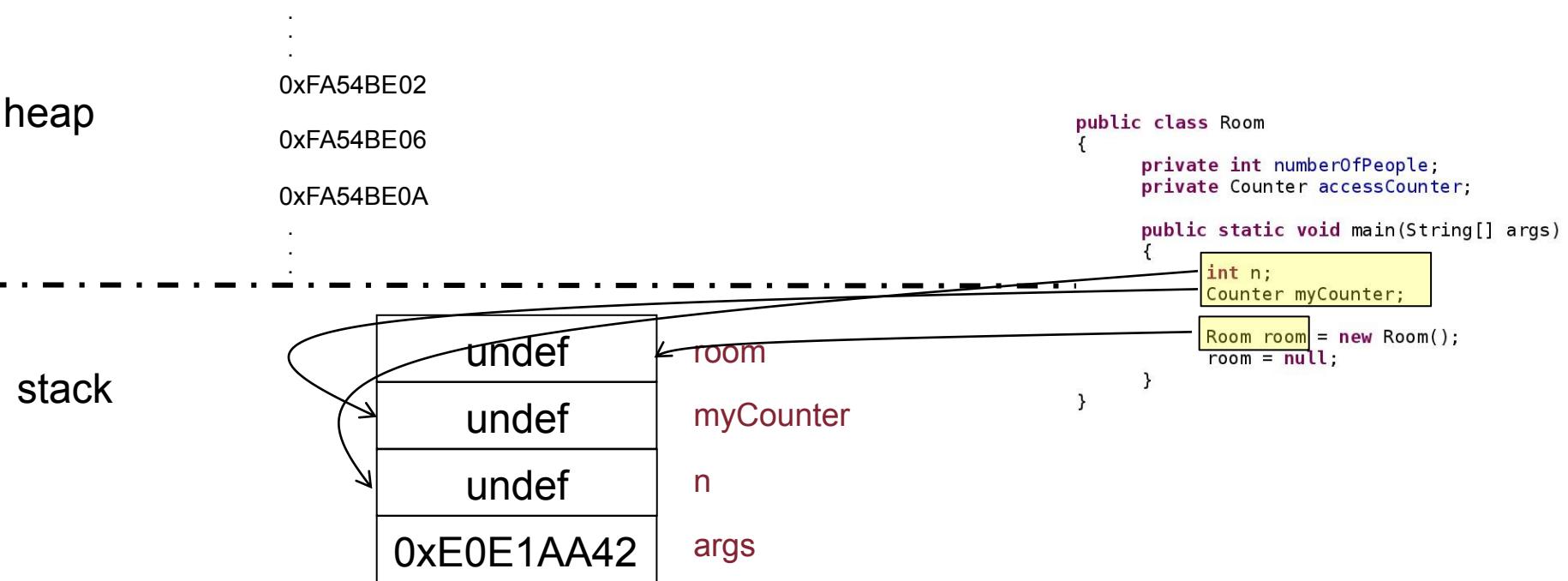
- **Dichiarazione:**
  - Menu menu = new Menu();
- **Creazione:**
  - Menu menu = new Menu();
- **Assegnazione:**
  - Menu menu = new Menu();

# Anatomia della memoria

Esistono due tipi di memoria: lo **heap** e lo **stack**

Sullo **stack** vanno le variabili **locali**

Sullo **heap** vanno le aree di memoria allocate per la creazione dinamica

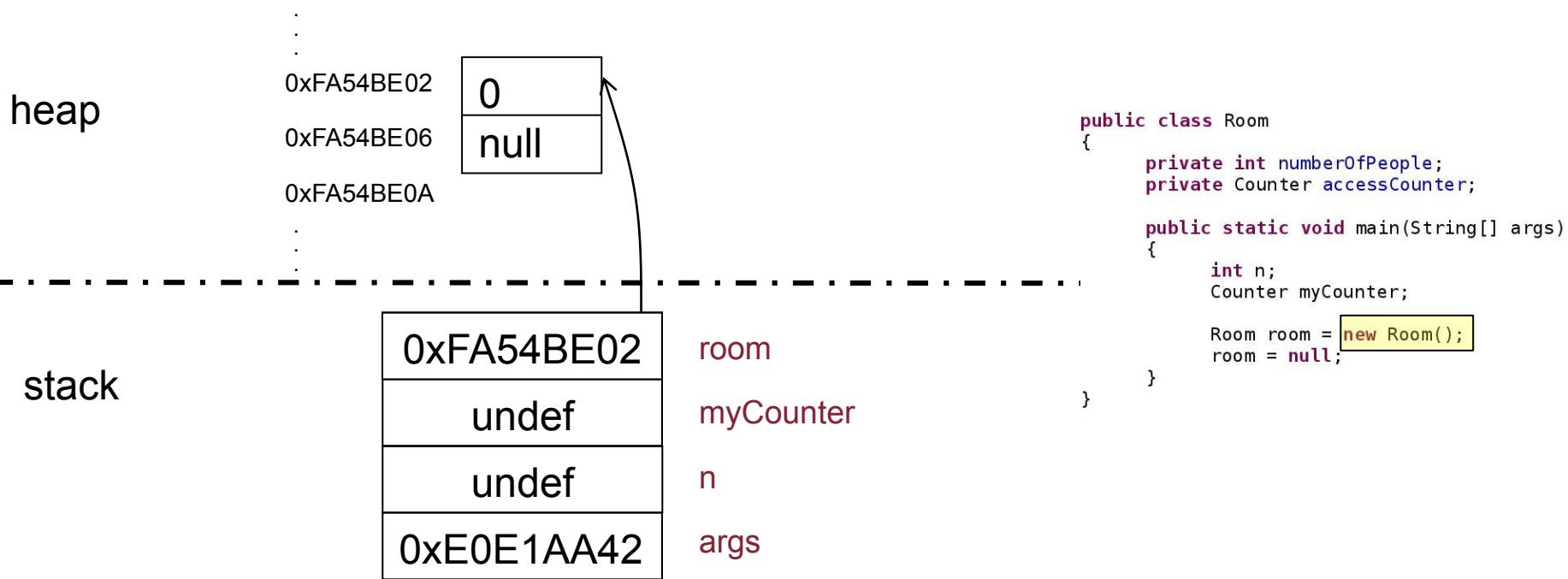


# Anatomia della memoria

Esistono due tipi di memoria: lo **heap** e lo **stack**

Sullo **stack** vanno le variabili **locali**

Sullo **heap** vanno le aree di memoria allocate per la creazione dinamica

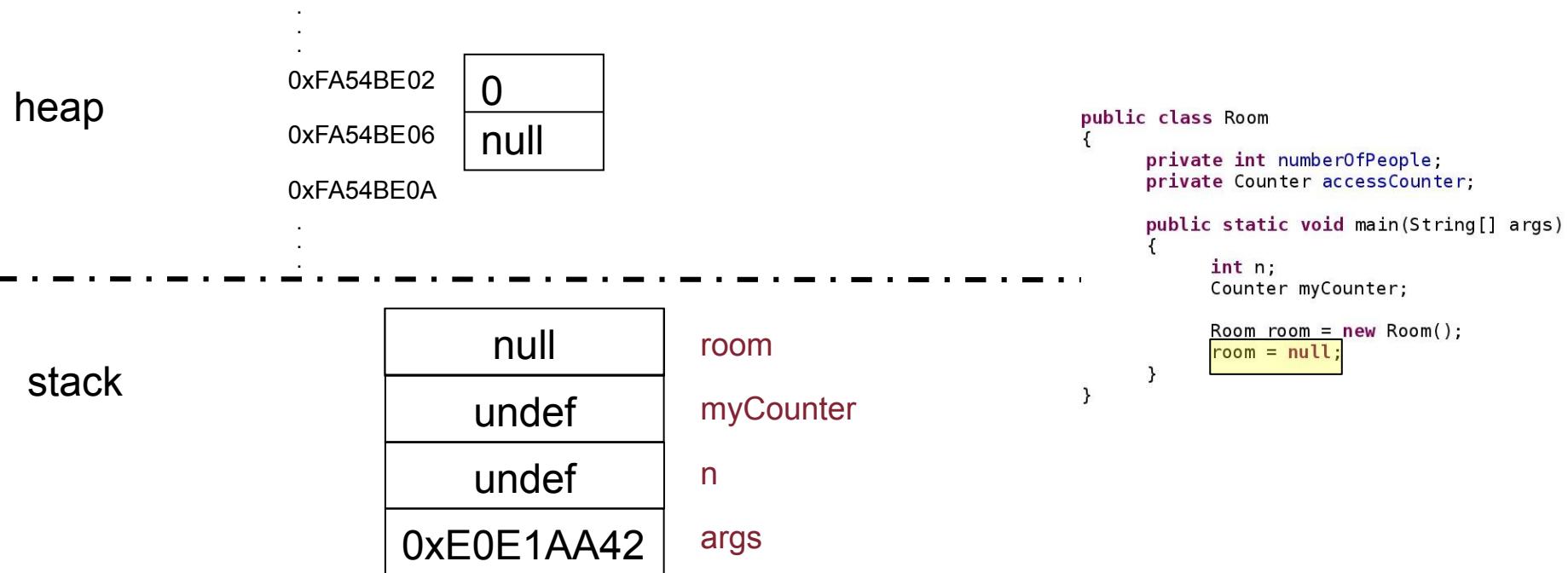


# Anatomia della memoria

Esistono due tipi di memoria: lo **heap** e lo **stack**

Sullo **stack** vanno le variabili **locali**

Sullo **heap** vanno le aree di memoria allocate per la creazione dinamica



## Campi di classe: la parola chiave static

- I campi di una classe possono essere dichiarati **static**
- Un campo **static** è relativo all'intera classe, **NON** al singolo oggetto istanziato
- Un campo **static** esiste in **una sola locazione di memoria**, allocata prima di qualsiasi oggetto della classe in una zona speciale di memoria nativa chiamata **MetaSpace**
- Viceversa, per ogni campo **non static** esiste **una locazione di memoria per ogni oggetto**, allocata a seguito dell'istruzione **new**

## Esempio

```
public class Room
{
    static private int totalNumberOfPeople;

    private int numberOfPeople;
    private Counter accessCounter;

    public static void main(String[] args)
    {
        int n;
        Counter myCounter;

        Room room1 = new Room();
        Room room2 = new Room();
    }
}
```

# Esempio

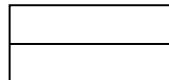
heap

0xFA54BE02

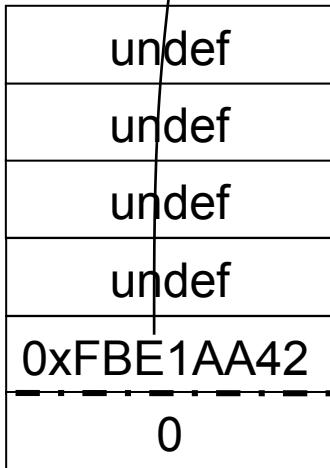
0xFA54BE06

0xFA54BE0A

:



stack



room2

room1

myCounter

n

args

totalNumberOfPeople

MetaSpace

```
public class Room
{
    static private int totalNumberOfPeople;

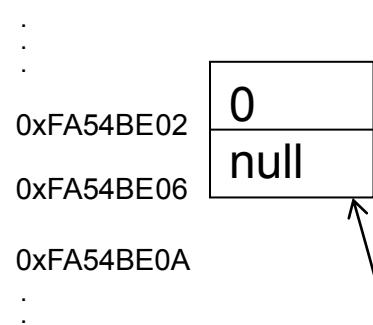
    private int numberOfPeople;
    private Counter accessCounter;

    public static void main(String[] args)
    {
        int n;
        Counter myCounter;

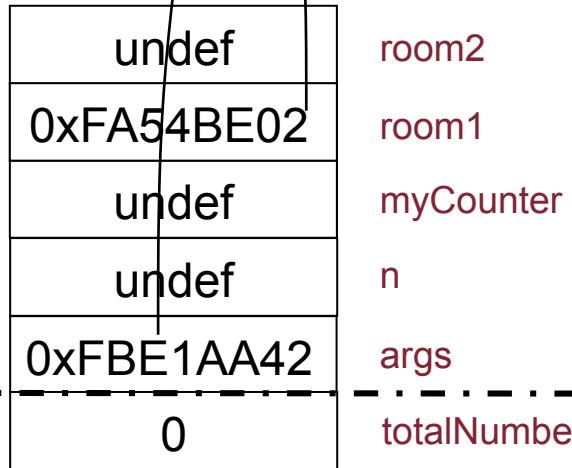
        Room room1 = new Room();
        Room room2 = new Room();
    }
}
```

# Esempio

heap



stack



MetaSpace

```
public class Room
{
    static private int totalNumberOfPeople;

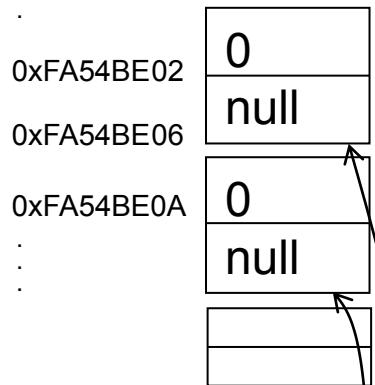
    private int numberOfPeople;
    private Counter accessCounter;

    public static void main(String[] args)
    {
        int n;
        Counter myCounter;

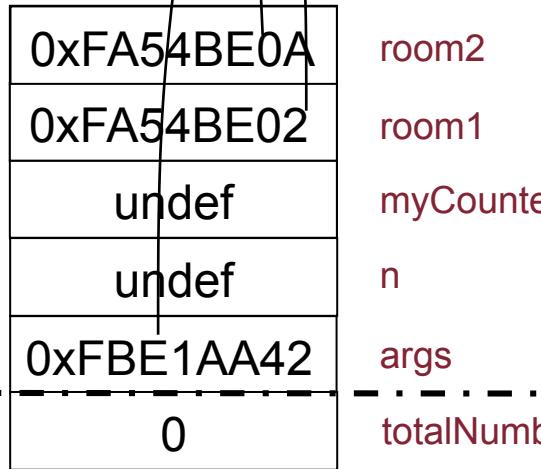
        Room room1 = new Room();
        Room room2 = new Room();
    }
}
```

# Esempio

heap



stack



MetaSpace

```
public class Room
{
    static private int totalNumberOfPeople;

    private int numberOfPeople;
    private Counter accessCounter;

    public static void main(String[] args)
    {
        int n;
        Counter myCounter;

        Room room1 = new Room();
        Room room2 = new Room();
    }
}
```

**Esercizio 1.1.2.** Disegnare lo stato della memoria (heap e stack) appena prima del termine dell'esecuzione del metodo `main` della classe seguente:

```
public class Tornello
{
    static private int passaggi;

    public void passa() { passaggi++; }
    public static int getPassaggi() { return passaggi; }

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k = 0; k < 10; k++) t2.passa();
        int g;
        String s = null;
        // fotografa qui lo stato della memoria
    }
}
```



```
public class Tornello {  
    static private int passaggi;  
  
    public void passa() {passaggi++;}  
    public static int getPassaggi() {return passaggi;}  
  
    public static void main(String[] args)  
{  
        Tornello t1 = new Tornello();  
        t1.passa();  
        Tornello t2 = new Tornello();  
        for (int k=0; k<10; k++) t2.passa();  
        int g;  
        String s=null;  
        // fotografa lo stato della memoria  
    }  
}
```

## Tornello



La prima cosa in assoluto è analizzare i campi statici

stack      heap

```
public class Tornello {  
    → static private int passaggi;  
  
    public void passa() {passaggi++;}  
    public static int getPassaggi() {return passaggi;}  
  
    public static void main(String[] args)  
    {  
        Tornello t1 = new Tornello();  
        t1.passa();  
        Tornello t2 = new Tornello();  
        for (int k=0; k<10; k++) t2.passa();  
        int g;  
        String s=null;  
        // fotografa lo stato della memoria  
    }  
}
```

## Tornello



stack      heap

passaggi=0

metaspace

```

public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args) {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



Analizziamo  
il main

stack      heap

main

passaggi=0

metaspace

```

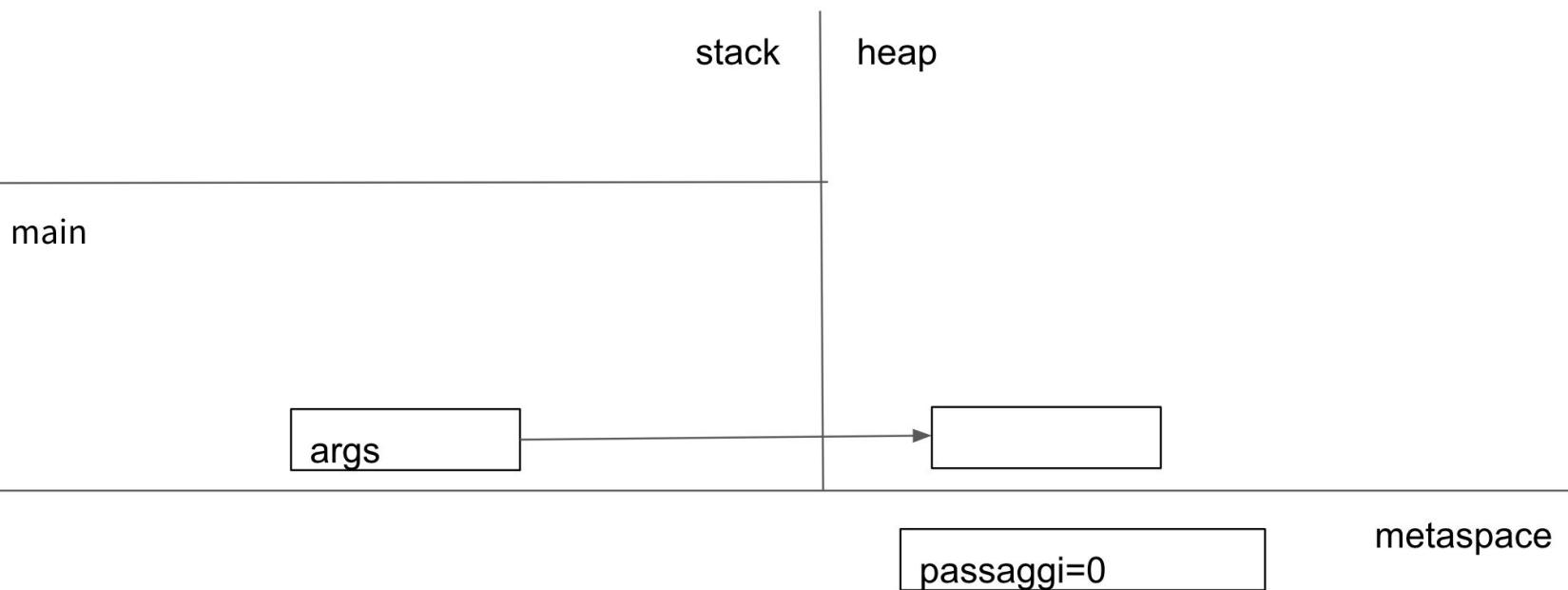
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    → public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

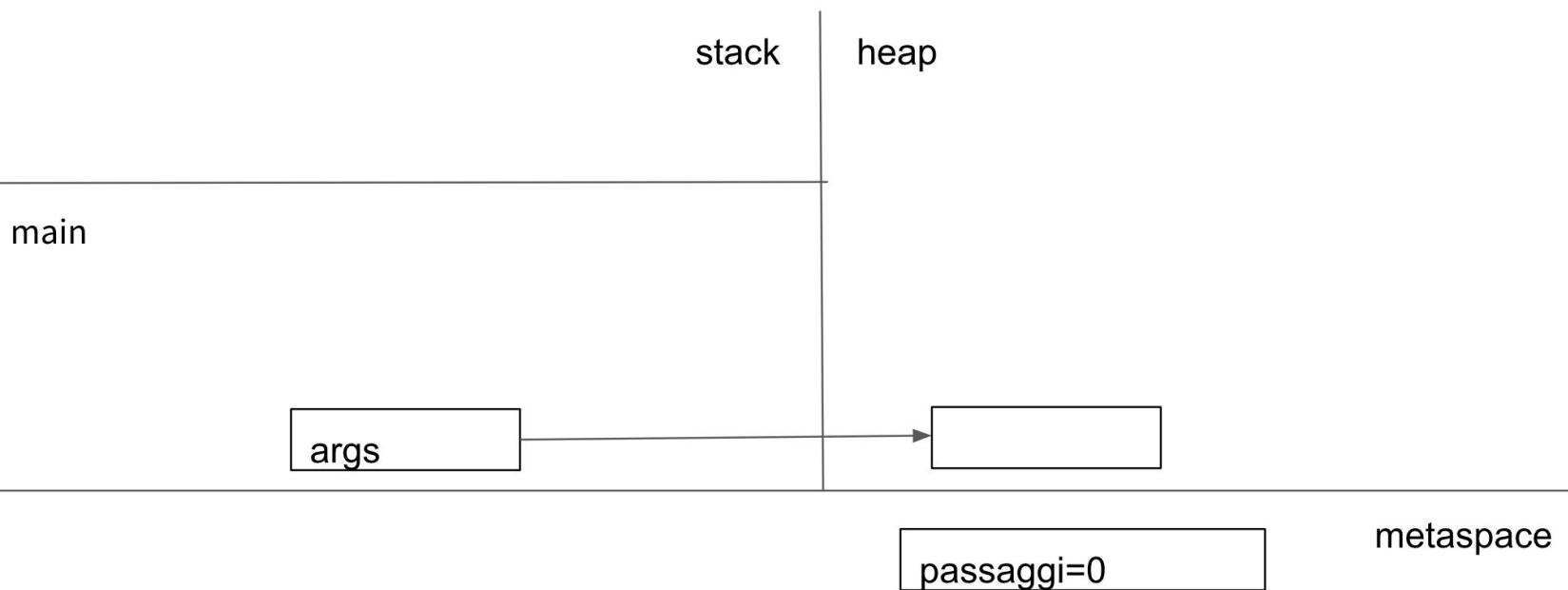
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    → public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

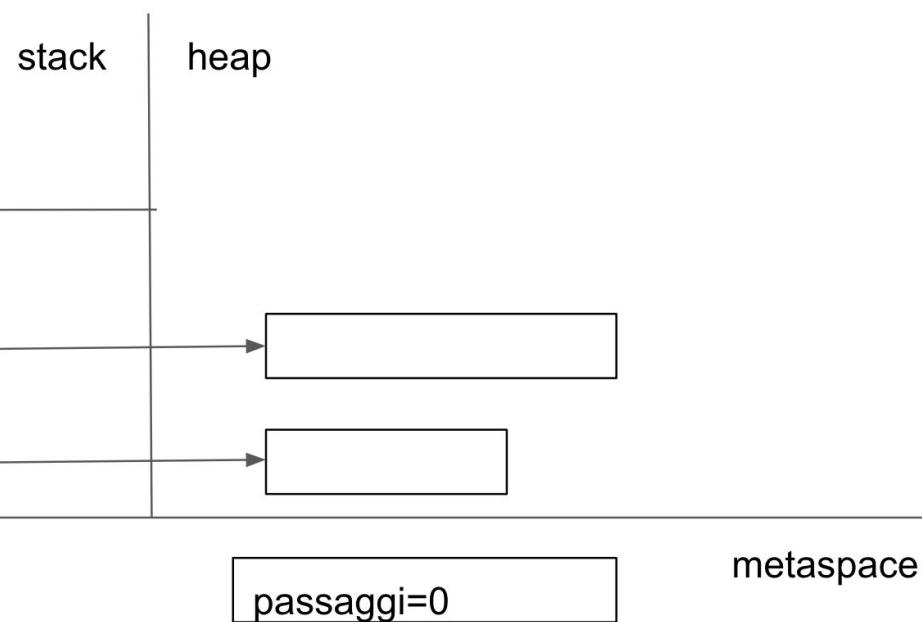
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

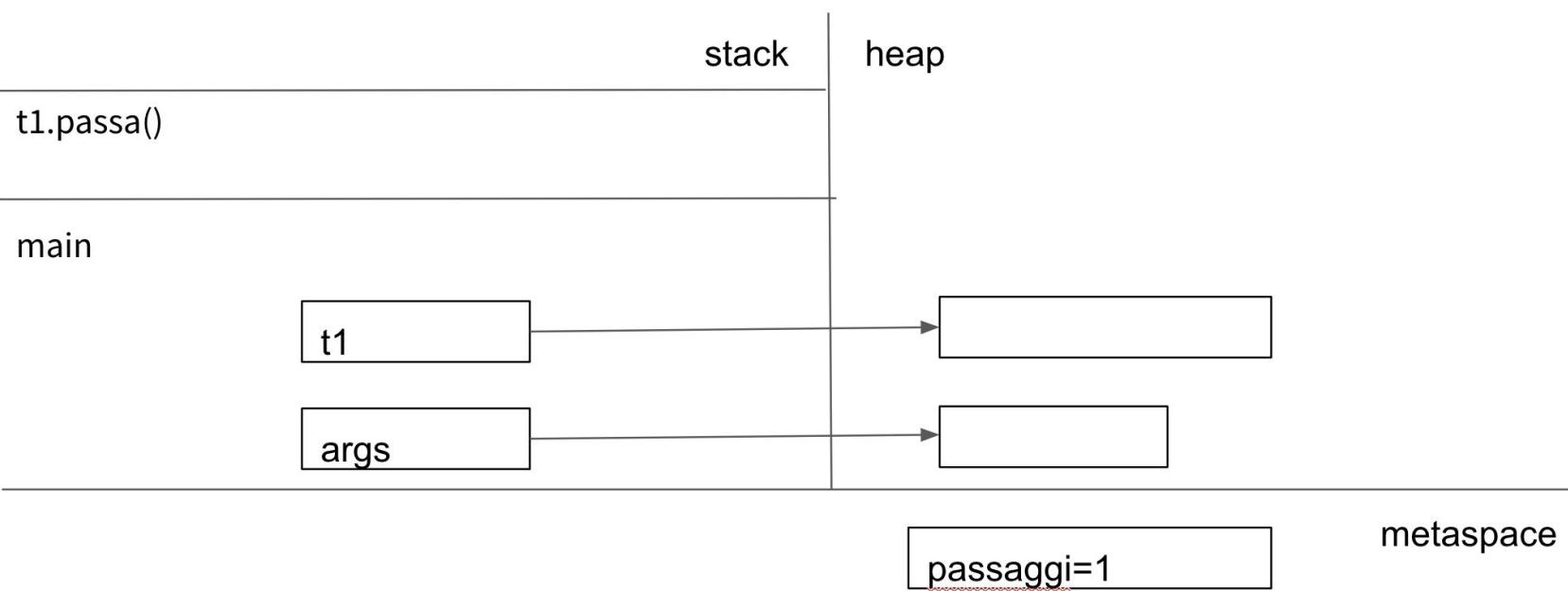
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

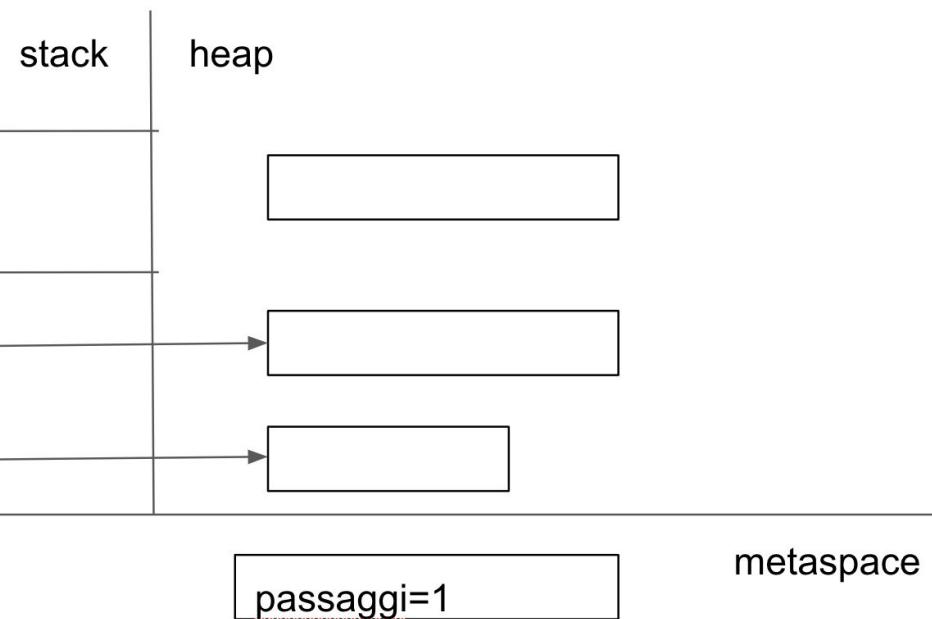
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

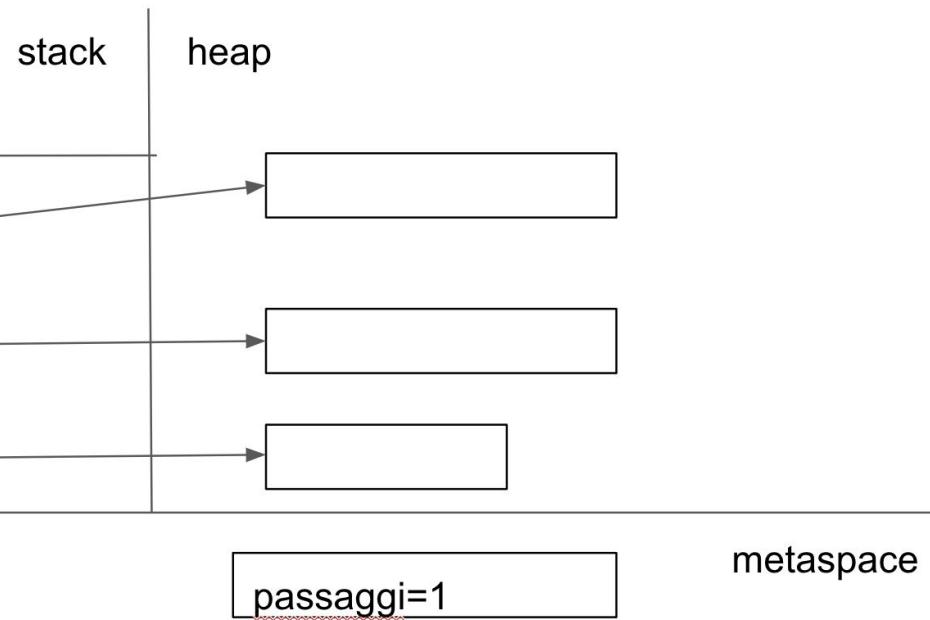
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

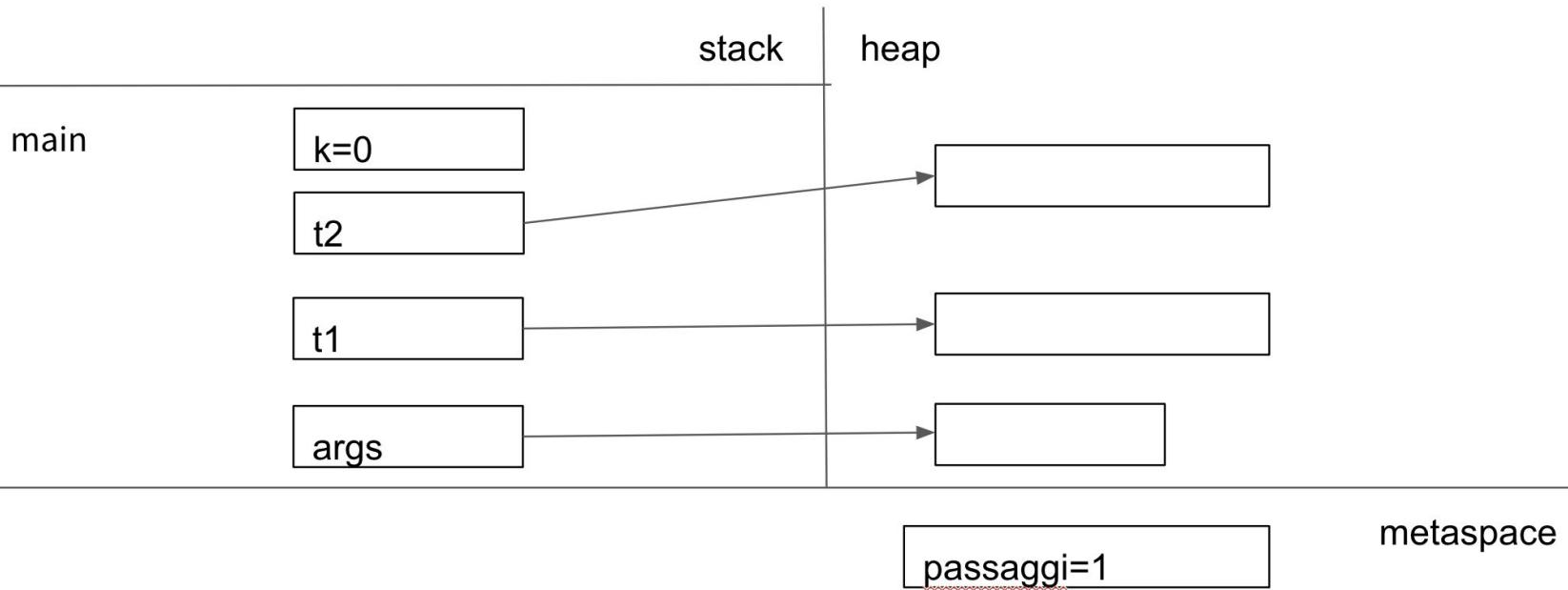
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

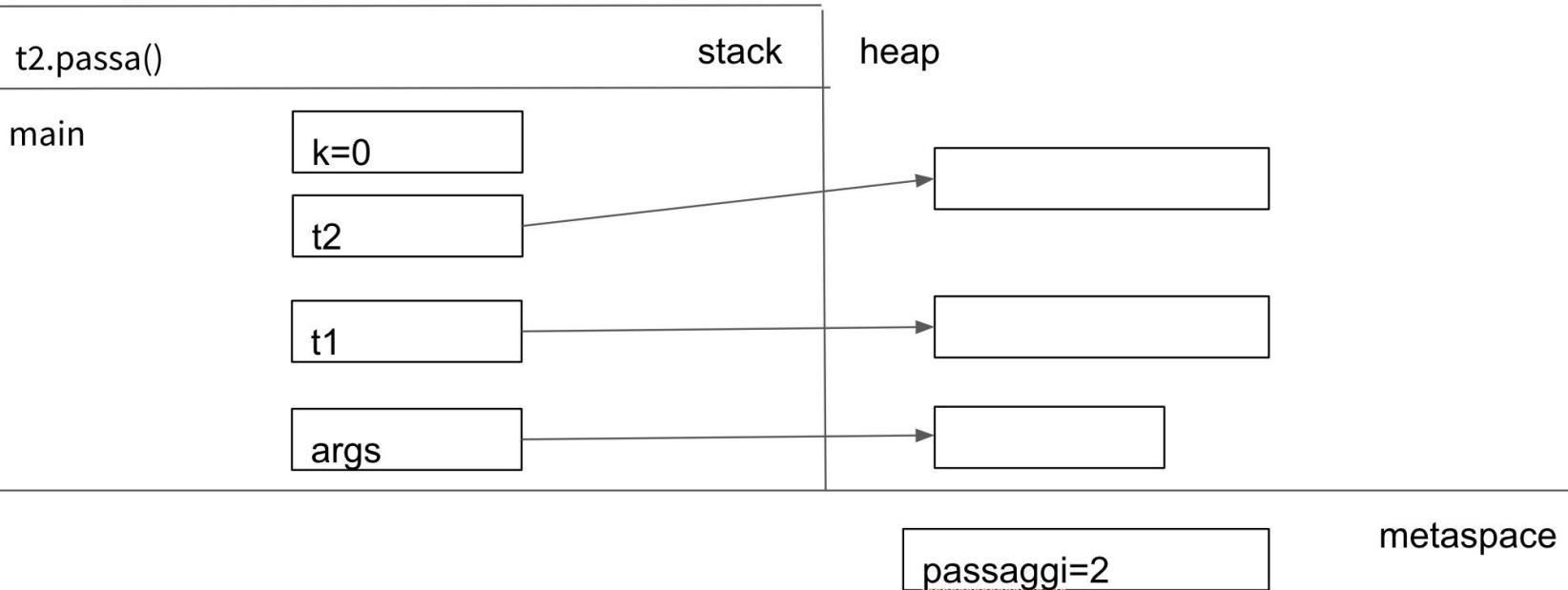
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

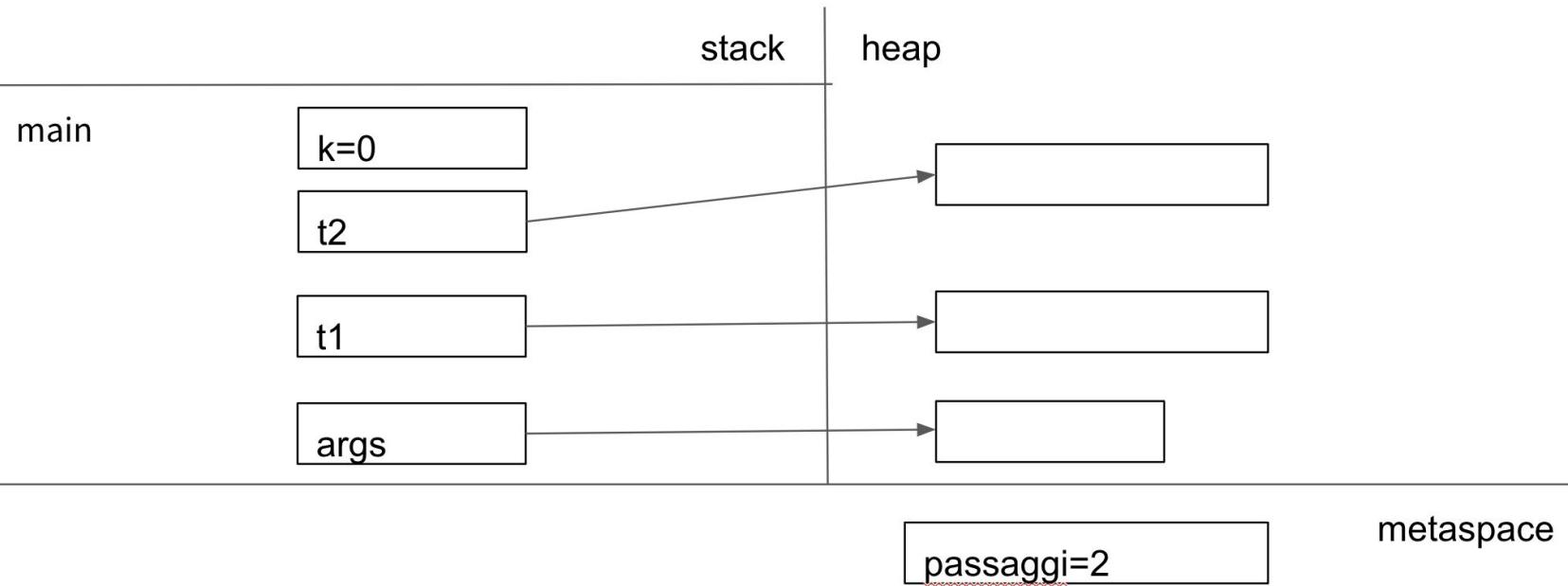
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

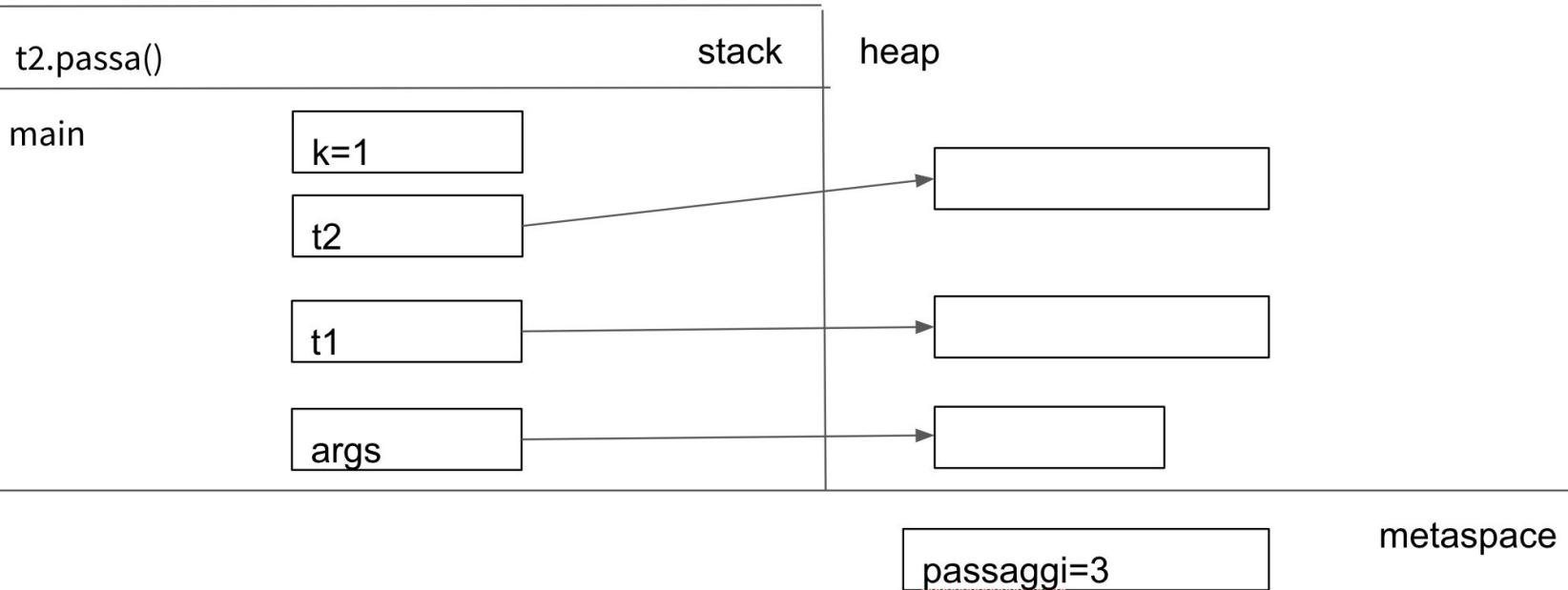
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello





```

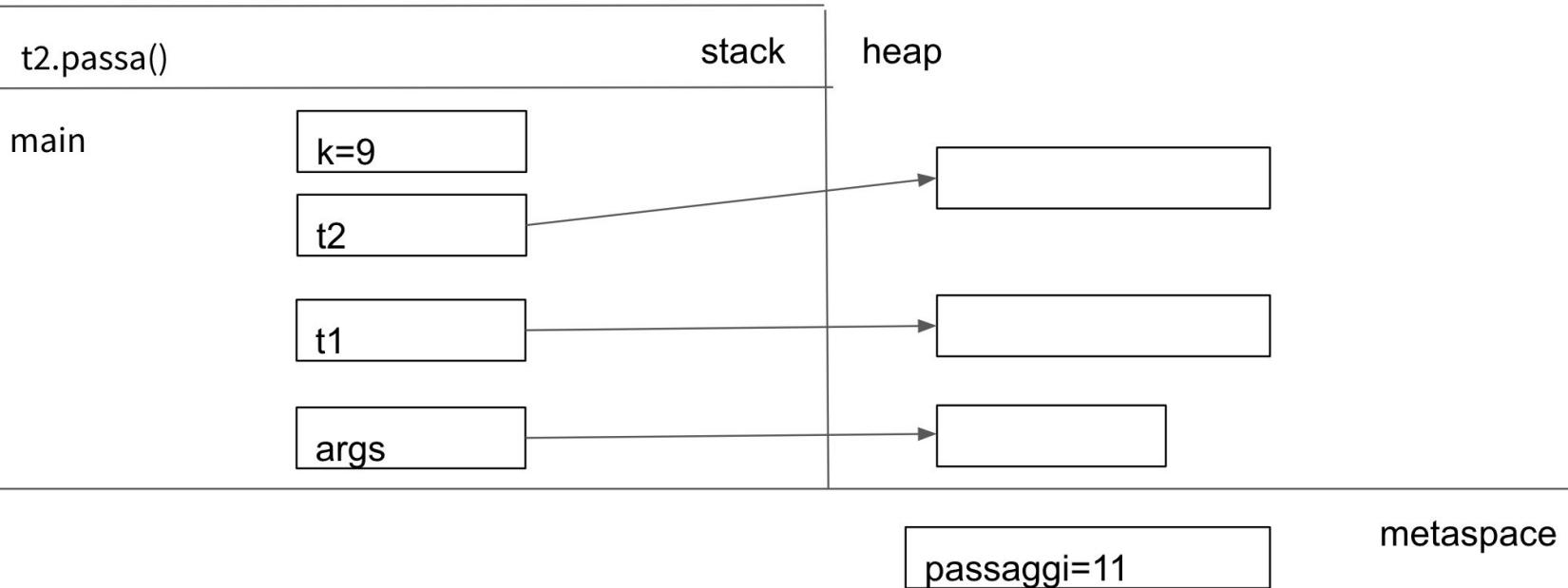
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

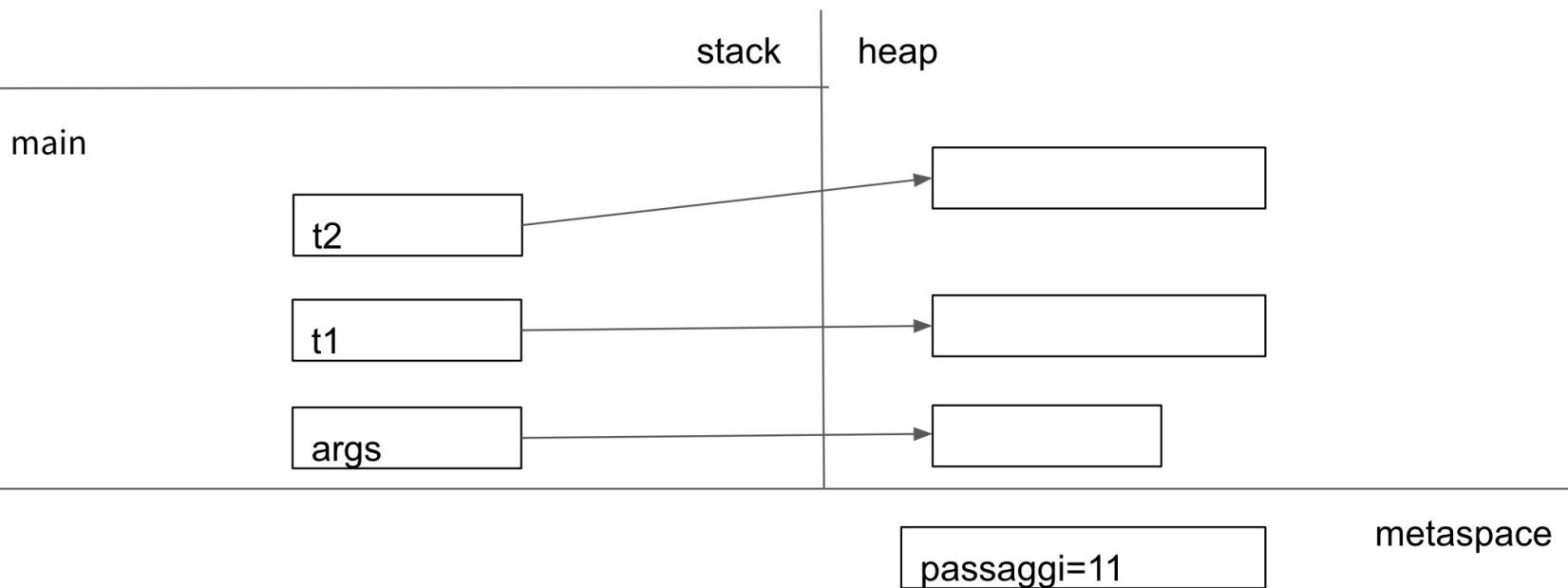
public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



```

public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



stack

heap

main

g undef

t2

t1

args

passaggi=11

metaspace

```

public class Tornello {
    static private int passaggi;

    public void passa() {passaggi++;}
    public static int getPassaggi() {return passaggi;}

    public static void main(String[] args)
    {
        Tornello t1 = new Tornello();
        t1.passa();
        Tornello t2 = new Tornello();
        for (int k=0; k<10; k++) t2.passa();
        int g;
        String s=null;
        // fotografa lo stato della memoria
    }
}

```

## Tornello



stack

heap

main

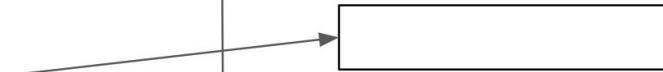
s null

g undef

t2

t1

args



passaggi=11

metaspace

## Esercizio: punti e segmenti

- Progettare una classe **Punto** per la rappresentazione di un punto nello spazio tridimensionale
- E una classe **Segmento** per rappresentare un segmento nello spazio tridimensionale
- Scrivere una **classe di test** che crei:
  - due oggetti della classe Punto con coordinate (1, 3, 8) e (4, 4, 7)
  - un oggetto della classe Segmento che rappresenti il segmento che unisce i due punti di cui sopra
- Raffigurare l'evoluzione dello stato della memoria, distinguendo tra **stack**, **heap** e **metaspace**

# Metodi statici

- I metodi **statici** sono **metodi di classe**
- **NON** hanno **accesso ai campi di istanza**
- Ma hanno **accesso ai campi di classe**

The diagram shows a Java code snippet with annotations. A vertical arrow on the left points upwards from the text 'accesso da metodo non statico a campo statico' to the line 'static private int numberOfInstances;'. Another curved arrow originates from the text 'accesso da metodo statico a campo statico' and points to the same line. The code itself is as follows:

```
public class ContaIstanze
{
    static private int numberOfInstances;

    public ContaIstanze()
    {
        numberOfInstances++;
    }

    static public void main(String[] args)
    {
        new ContaIstanze();
        new ContaIstanze();
        new ContaIstanze();

        System.out.println("Numero di istanze create finora: "+numberOfInstances);
    }
}
```

## Lettura dell'input da console

- Si effettua con la classe `java.util.Scanner`
- Costruita passando al costruttore lo stream di input (`System.in` di tipo `java.io.InputStream`)

```
public class ChatBotNonCosìInterattivo
{
    public static void main(String[] args)
    {
        // crea uno Scanner per ottenere l'input da console
        java.util.Scanner input = new java.util.Scanner(System.in);

        System.out.println("Come ti chiami?");

        // legge i caratteri digitati finche' non viene inserito
        // il carattere di nuova riga (l'utente preme invio)
        String nome = input.nextLine();

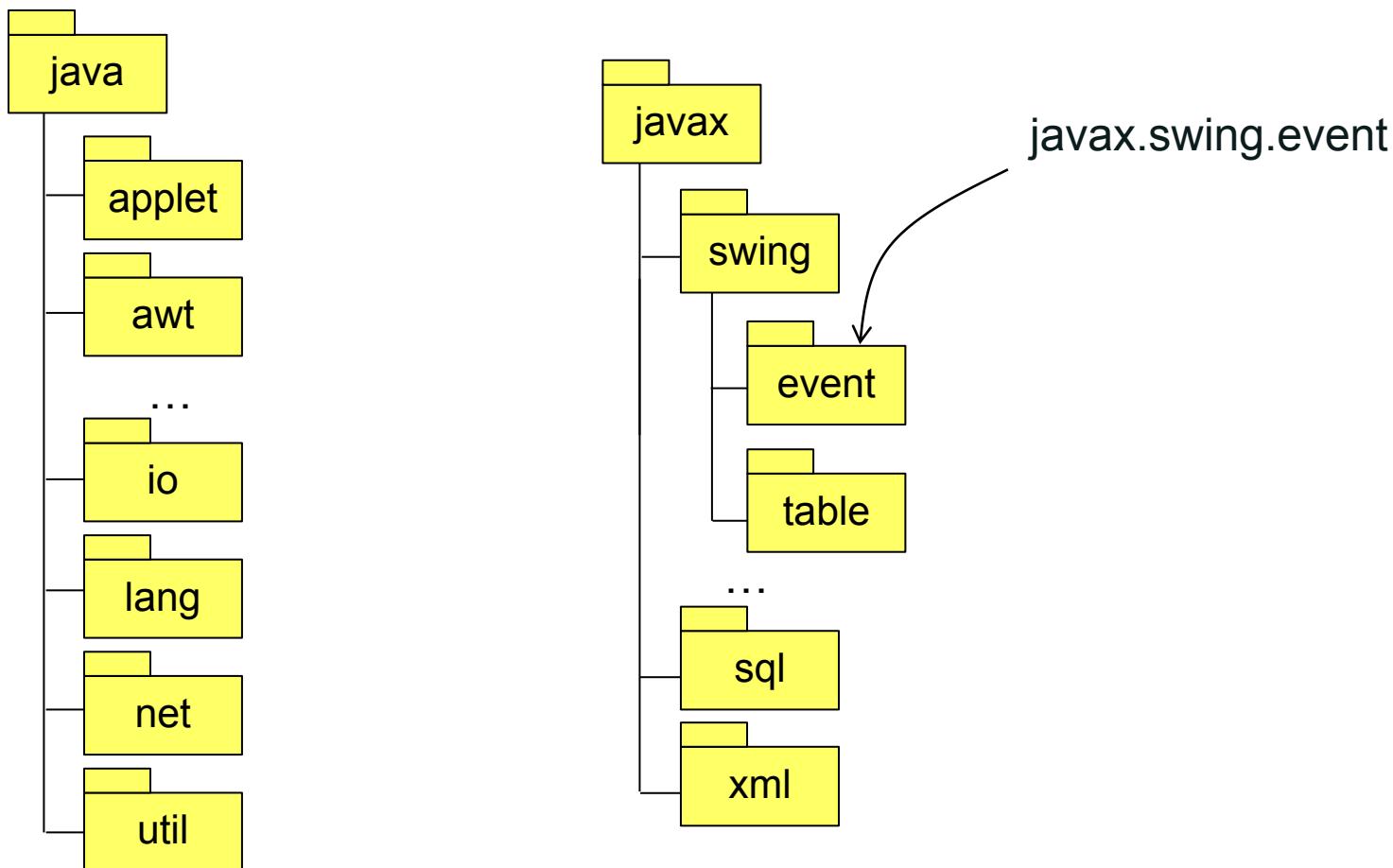
        System.out.println("Ciao "+nome+"!");
    }
}
```

# Package

- Le classi vengono inserite (categorizzate) in collezioni dette **package**
- Ogni package racchiude classi con **funzionalità correlate**
- Quando si utilizza una classe è necessario **specificarne il package** (come per **Scanner**, che appartiene al package `java.util`)
- Le classi che abbiamo usato finora (es. **System**, **String**) appartengono al package **speciale** `java.lang`
  - Questo package non deve essere specificato

# Package standard

- Le API (Application Programming Interface) di Java sono organizzate in numerosi package



# La dichiarazione import

- Per evitare di specificare il package di una classe ogni volta che viene usata, è sufficiente **importare la classe**

```
import java.util.Scanner;
```

```
public class ChatBotNonCosìInterattivo
{
    public static void main(String[] args)
    {
        // crea uno Scanner per ottenere l'input da console
        Scanner input = new Scanner(System.in);

        System.out.println("Come ti chiami?");

        // leggere i caratteri digitati finche' non viene inserito
        // il carattere di nuova riga (l'utente preme invio)
        String nome = input.nextLine();

        System.out.println("Ciao " + nome + " !");
    }
}
```

# La dichiarazione import

- Per evitare di specificare il package di una classe ogni volta che viene usata, è sufficiente **importare** la classe
- O l'intero **package**:

```
import java.util.*;
```

**Attenzione:** non è ricorsivo!

```
public class ChatBotNonCosiInterattivo
{
    public static void main(String[] args)
    {
        // crea uno Scanner per ottenere l'input da console
        Scanner input = new Scanner(System.in);

        System.out.println("Come ti chiami?");

        // leggere i caratteri digitati finche' non viene inserito
        // il carattere di nuova riga (l'utente preme invio)
        String nome = input.nextLine();

        System.out.println("Ciao "+nome+" !");
    }
}
```

# Creazione di nuovi package

- I **package** sono rappresentati fisicamente da **cartelle** (**String.class** si trova sotto **java/lang/**)
- Una classe può essere inserita in un determinato **package** semplicemente
  - specificandolo all'inizio del file (parola chiave **package**)
  - posizionando il file nella corretta sottocartella
- **Eclipse fa tutto questo per voi!**

# Esempio: la classe Triangolo

```
package it.navigli.geometry;  
  
/**  
 * La figura geometrica triangolo  
 * @author navigli  
 */  
  
public class Triangolo  
{  
    /**  
     * Base del triangolo  
     */  
    private double base;  
  
    /**  
     * Altezza del triangolo  
     */  
    private double altezza;  
  
    /**  
     * Costruttore del triangolo  
     * @param base base del triangolo  
     * @param altezza altezza del triangolo  
     */  
    public Triangolo(double base, double altezza)  
    {  
        this.base = base;  
        this.altezza = altezza;  
    }  
  
    /**  
     * Restituisce l'area della figura  
     * @return l'area  
     */  
    public double getArea()  
    {  
        return base*altezza/2.0;  
    }  
}
```

Triangolo.java si deve trovare nella cartella it/navigli/geometry

Commento Javadoc per la classe

Javadoc per un campo

Javadoc per il costruttore

Parametri Javadoc per il costruttore

Javadoc per un metodo

Riferimento all' oggetto costruito

Javadoc per il valore restituito

## Credits

Le slide di questo corso sono il frutto di una personale rielaborazione delle slide del Prof. Navigli.

In aggiunta, le slide sono state revisionate dagli studenti borsisti della Facoltà di Ingegneria Informatica, Informatica e Statistica: Mario Marra e Paolo Straniero.