LEZIONE 6
CLASS DIAGRAMS 1:
Classi, Oggetti ed Information Hiding

Ingegneria del Software e Progettazione Web
Università degli Studi di Tor Vergata - Roma

Guglielmo De Angelis
guglielmo.deangelis@isti.cnr.it

object orientation

object orientation

i concetti fondamentali del mondo Object Oriented sono:

fondamentali

- classe
- istanza
- metodo

} oggetto

- operazione
- messaggio
- incapsulamento
- ereditarietà
- polimorfismo

} necessari per avere O.O.

classe

- modella
  - una famiglia di entità del dominio di applicazione
  - elemento che non fa parte del nel dominio ma che è introdotto durante il processo di sviluppo
    - aumentare modificabilità/estensibilità del sistema, realizzare interfacce grafiche, … …
- in entrambi i casi include la definizione di :
  - le proprietà (attributi)
  - il comportamento (operazioni)
- raggruppa un insieme coeso di entità
  - coesione vs. accoppiamento
    - insieme coeso: tutti gli elementi modellano aspetti utili al raggiungimento dello stesso fine/funzionalità
    - mammifero, autoveicolo, grafo... OK!!
    - (in una biblioteca) l'insieme dei libri con inclusi riferimenti anagrafici degli autori … KO!!
- MAI interpretare una classe un “ *functoid* ”
  - funzione procedurale *travestita* da classe

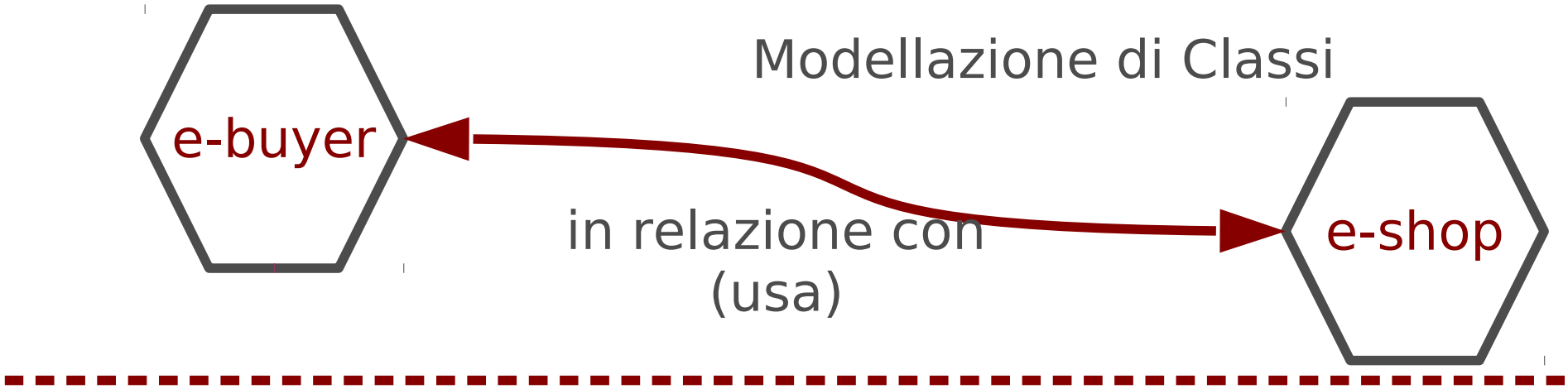
classe

- un oggetto si relaziona ad una classe allo stesso modo di come un dato si relaziona ad un tipo (nei ling. di progr.)
- tutti gli *oggetti* (i.e. istanze) afferenti una classe
  - condividono lo stesso insieme :
    - di comportamenti
    - di proprietà
    - di relazioni
  - (in generale) differiscono nei valori delle proprietà

istanza

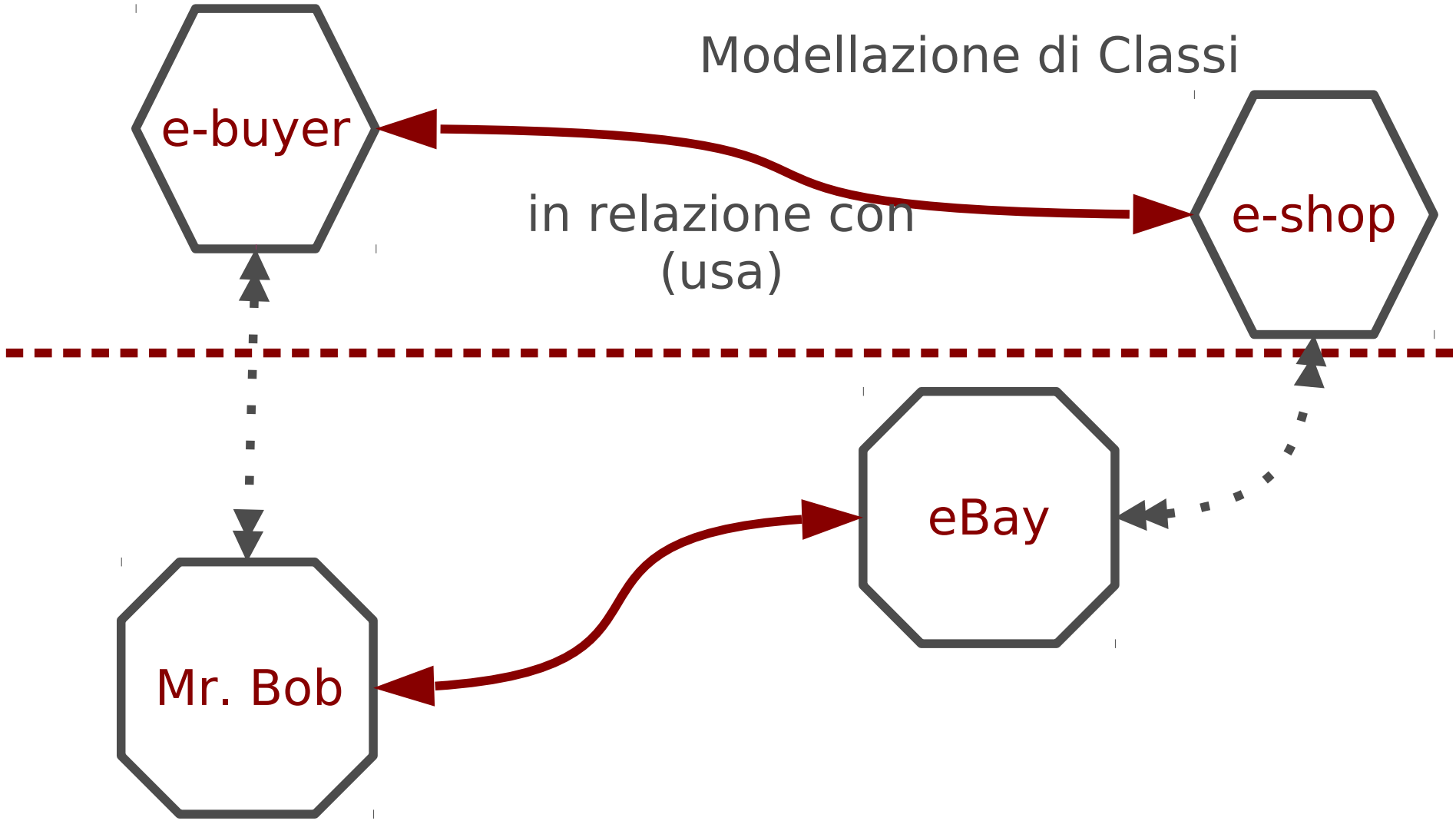
- rappresenta un “elemento del mondo reale” che si sta modellando
- ha associato un tipo (definito dalla classe)
- può modellare un'entità fisica, concettuale o software
  - il mio gatto, l'auto di mio padre, il grafo della rete ferroviaria regionale ...
- manifestazione concreta di un’astrazione
- ha un’identità ben definita e incapsula uno stato e un comportamento
- in ogni istante, lo stato è definito dall'insieme dei valori delle proprietà definite nella classe

classi VS. oggetti

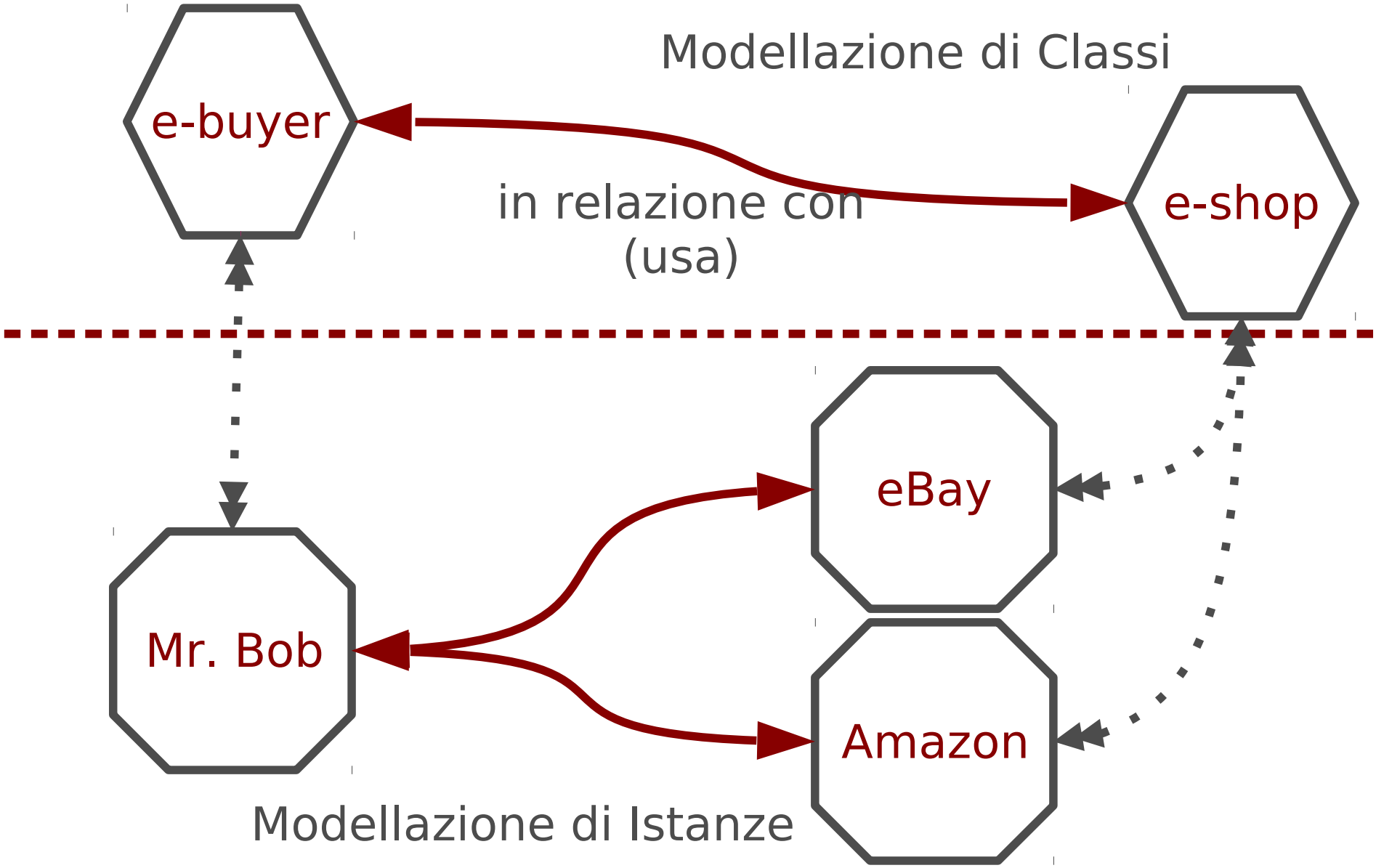


Modellazione di Istanze

classi VS. oggetti



classi VS. oggetti



dinamica di un sistema

- in un approccio **procedurale**
  - esiste un “main” che coordina il comportamento del sistema in base allo scheletro del diagramma di flusso codificato nelle istruzioni
  - il “main” gestisce direttamente i dati del sistema, occupandosi esplicitamente della loro transizione tra le varie funzioni
- in un approccio **object oriented**
  - ogni oggetto ha i suoi “dati” (valori per gli attributi di istanza)
    - gli attributi di istanza sono allocati ed inizializzati alla creazione dell’oggetto,
    - l'istanza li mantiene anche dopo che i metodi sono stati eseguiti (a differenza di quanto accade per le tradizionali invocazioni a funzione)
    - tutti i dati sono mantenuti fino alla distruzione esplicita/implicita dell’oggetto (vedere slide successive su distruzione)
  - più oggetti interagiscono tra loro per “generare il comportamento” del sistema
  - l'interazione comporta lo scambio di *messaggi* tra oggetti diversi
  - l'interazione può produrre transizioni di stato negli oggetti coinvolti

operazione, metodo, messaggio

- operazione: specifica di un "*segnatura*" (prototipo) che un oggetto mette a disposizione di altri oggetti

- metodo: implementazione vera e propria dell'operazione di un oggetto

- messaggio: è la richiesta a run-time di invocazione di un metodo fornito da un oggetto $\mathcal{B}$ da parte di un oggetto $\mathcal{A}$

la famiglia di diagrammi UML

- structure diagrams
  - class diagrams
  - object diagrams
  - component diagrams
  - deployment diagrams
  - composite structure diagrams
  - package diagrams
- behavior diagrams
  - state machine diagrams
  - activity diagrams
  - use case diagrams
  - interaction diagrams
    - sequence diagrams
    - communication diagrams
    - interaction overview diagrams
    - timing diagrams

class diagrams

- struttura statica del sistema:
  - ha una rappresentazione logica a grafo
  - nodi + relazioni

class diagrams

- struttura statica del sistema:
  - nodi + relazioni
- un nodo modella una “classe” che rappresenta:
  - una entità del dominio
  - elementi che non fanno parte del dominio ma utili nell'ingegnerizzazione del sistema
- una classe è caratterizzata
  - da un nome
  - degli attributi
  - delle operazioni sugli attributi
- sono lo stesso concetto in O.O. (introdotto nelle precedenti slide)
  - (semplificando) rappresentano un tipo di dato: l’insieme dei “campi” e dei valori che ognuno di essi ammette && l'insieme di operazioni che ne definiscono i modi di interazione
  - dipendentemente dalla “vista” di riferimento corrispondono ad una implementazione dell'entità

class diagrams

- struttura statica del sistema:
  - nodi + relazioni
- le relazioni di base in un class diagrams
  - semplificando : corrispondono alla definizione delle possibili interazioni tra le classi di un modello
    - una *relazione* tra una classe A ed una classe B significa che A è a conoscenza di B e (in qualche *modo*) può interagirvi
    - il tipo di relazione definisce il *modo* di interazione
    - l'assenza di relazioni nella classe implica l'isolamento di quella classe:
      - é impossibilitata ad interagire con le altre classi

class diagrams

- struttura statica del sistema:
  - nodi + relazioni
- le relazioni di base in un class diagrams
  - semplificando : corrispondono alla definizione delle possibili interazioni tra le classi di un modello
    - una *relazione* tra una classe A ed una classe B significa che (in qualche *modo*) A può interagire con B
    - il tipo di relazione definisce il *modo* di interazione

association
aggregation
composition

dependency
generalization
interfaceRealization

realization

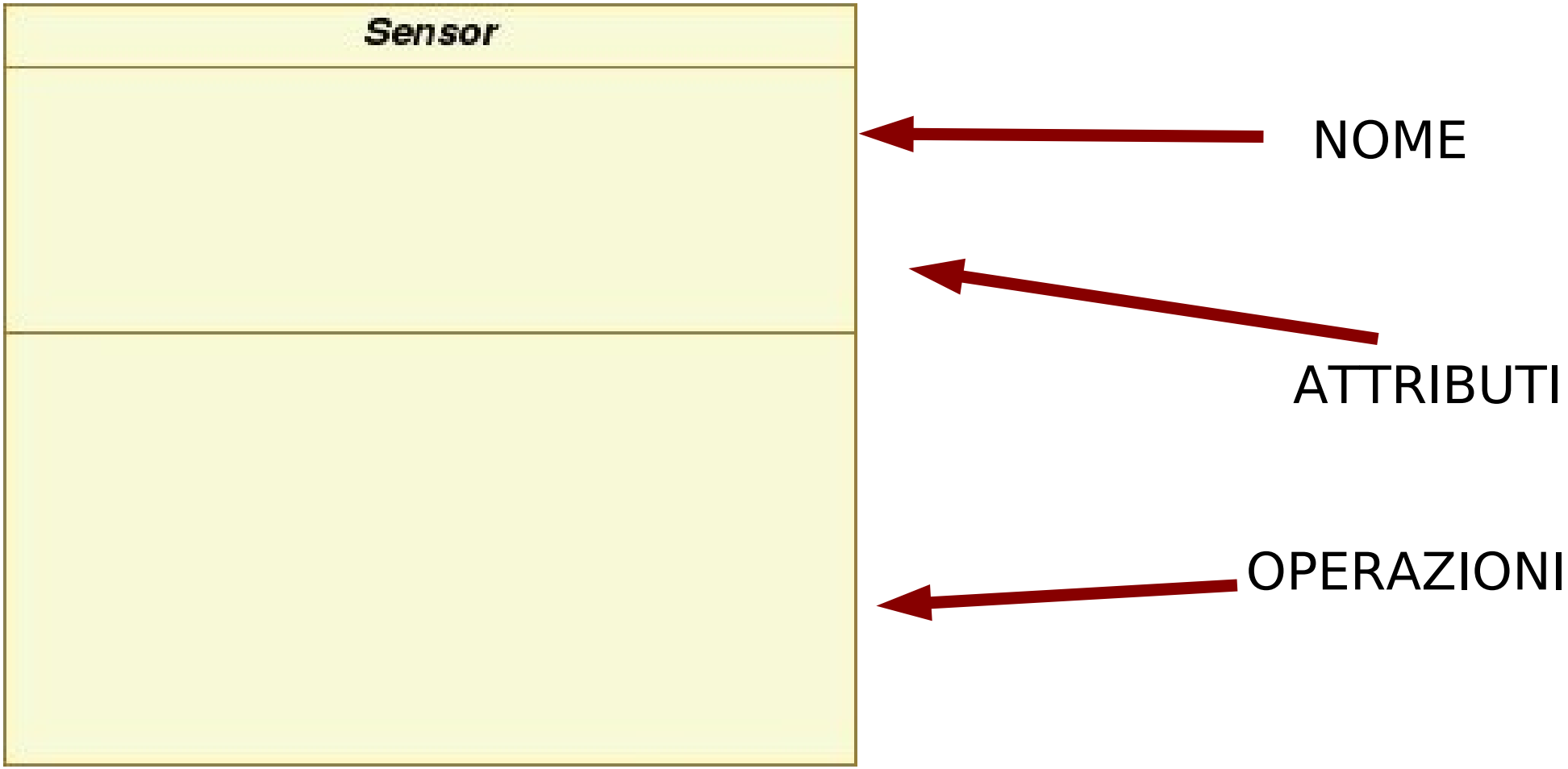
class diagrams – riassumendo

- mostra un insieme di classi (ed *interfacce*) e le relazioni tra loro
  - dipendenza, associazione, aggregazione, composizione, generalizzazione
- può essere visto come un grafo dove i nodi sono classi/interfacce e gli archi relazioni
- possono contenere anche package o sottosistemi (usati per raggruppare elementi)

class diagrams – riassumendo

- modella la struttura statica di una applicazione
  - elementi specificati e/o composti a design-time
- si usa per modellare:
  - gli elementi di una applicazione
    - una classe è l'astrazione di un elemento nel dominio del problema (vocabolario)
    - elementi che non fanno parte del dominio ma utili nell'ingegnerizzazione del sistema
  - semplici collaborazioni
    - una classe non vive da sola ma si relaziona con altre al fine di fornire, "cooperativamente", un comportamento complesso

struttura di una classe – UML (1)

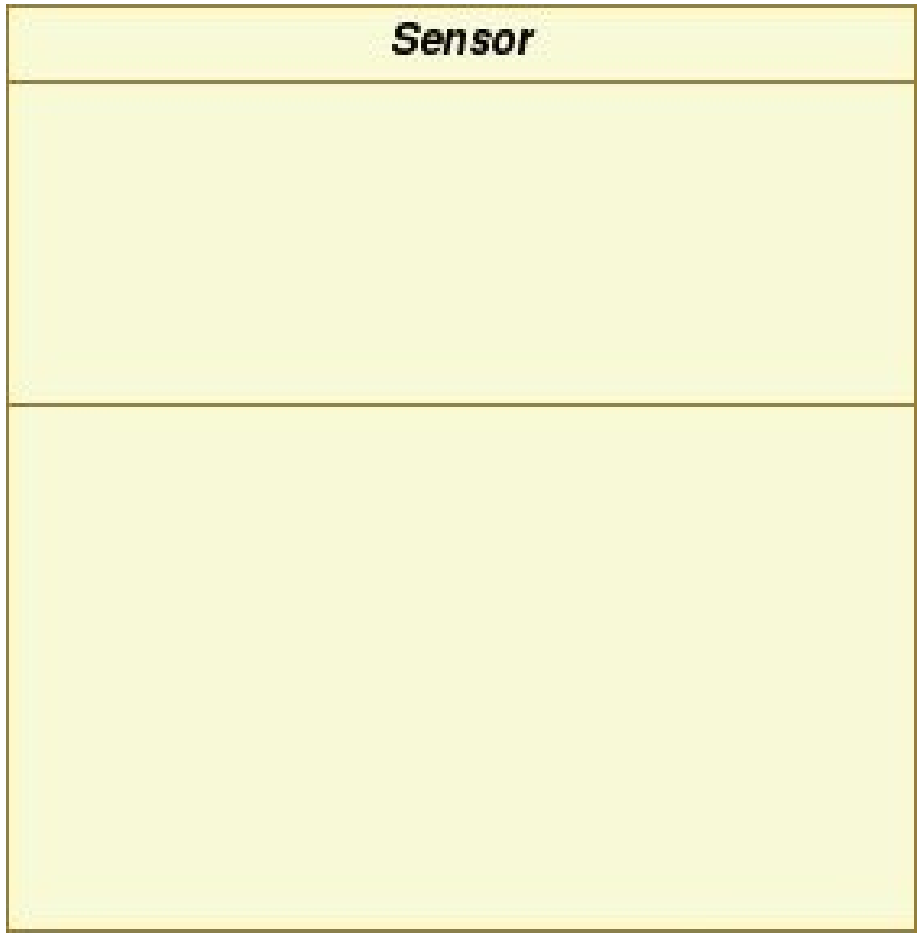


compartimento nome

- definisce il nome di un’entità
- stringa di testo
  - stringa contenente il solo nome della classe (i.e. "simple name")
  - definizione completa
    - Java: package + “.” + nome_classe
    - UML: prefisso + “::” + nome_classe
- convenzione
  - lettera iniziale maiuscola

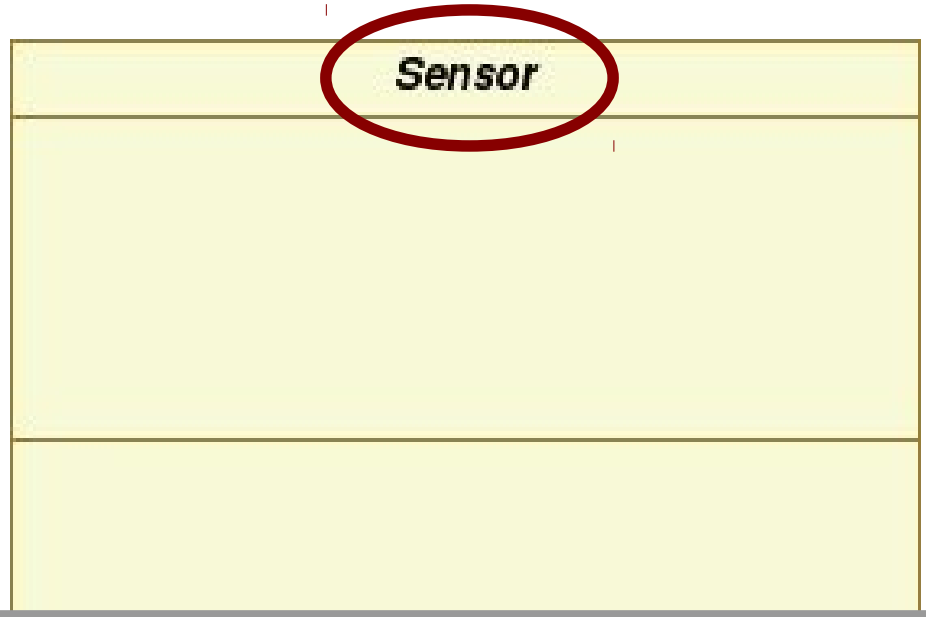
compartimento nome – UML && Java

```
public abstract class Sensor {
   /** Corpo della
     *     classe
   */
}
```



compartimento nome – UML && Java

```
public abstract class Sensor {
   /** Corpo della
     *     classe
   */
}
```



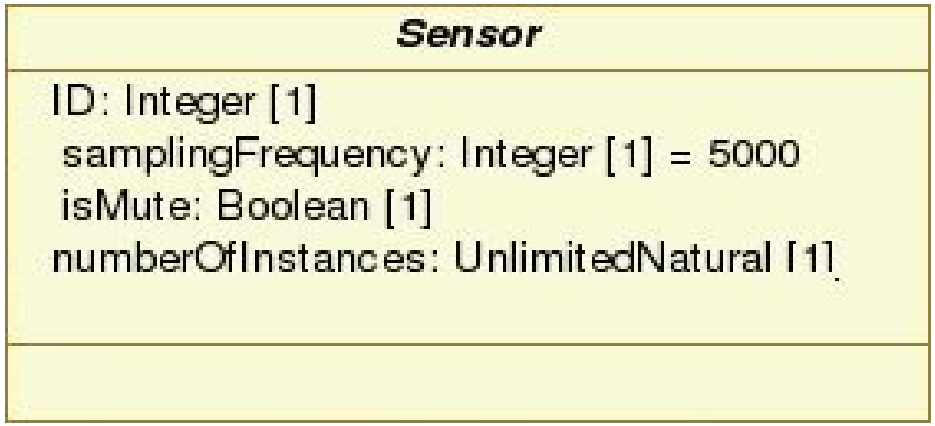
per il momento notare che se il nome della classe in UML è in *corsivo* allora in Java è richiesto il modificatore "`abstract`" e viceversa … approfondiremo il significato di questo modificatore nella prossima lezione

comparto attributi

- modella le proprietà di una classe
  - per ogni attributo, l'insieme di valori che può assumere e delle operazioni interne/esterne su di esso definite (tipo)
- le proprietà sono condivise tra tutti gli oggetti appartenenti a quella classe
  - tutte le istanze che hanno quella classe come tipo
  - (in generale) i valori non sono condivisi tra le istanze
- nel nostro esempio:
  - un sensore è caratterizzato da
    - identificativo (univoco)
    - frequenza di rilevamento della misura
    - modalità *standby*

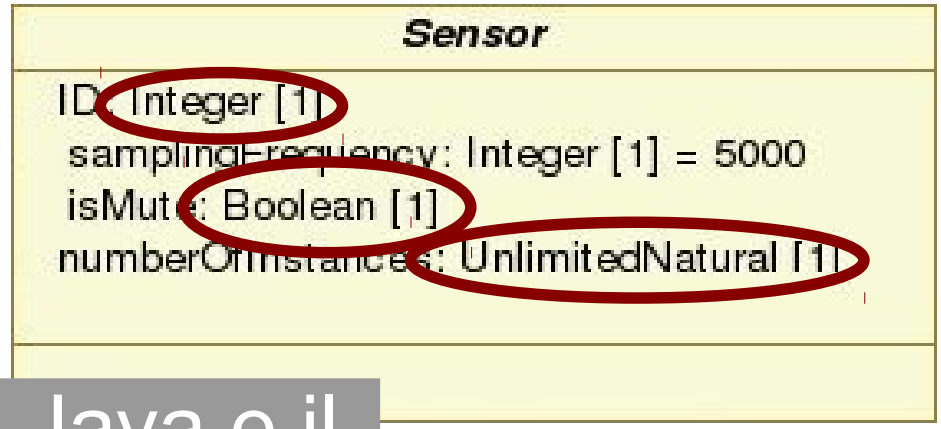
comparto attributi – UML && Java

```
public class Sensor {
   int ID;
   int samplingFrequency = 5000;
   boolean isMute;
   int numberOfInstrances;
}
```



comparto attributi – UML && Java

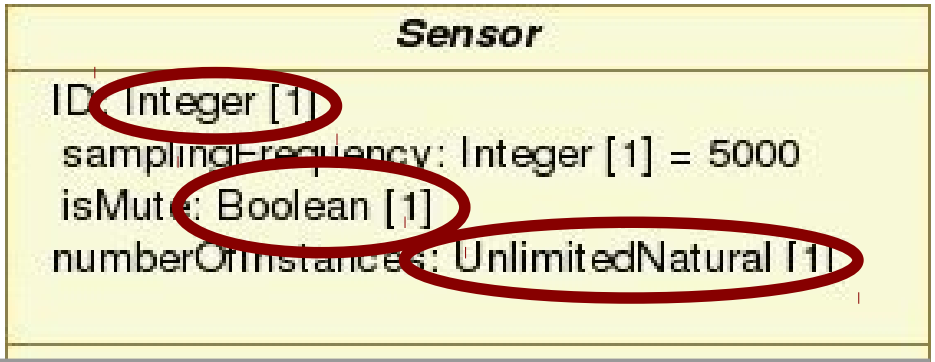
```
public class Sensor {
  int ID;
  int samplingFrequency = 5000;
  boolean isMute;
  int numberOfInstrances;
}
```



Attenzione!!! Tra il codice Java e il modello UML non c'è sempre una corrispondenza univoca e/o esatta!

comparto attributi – UML && Java

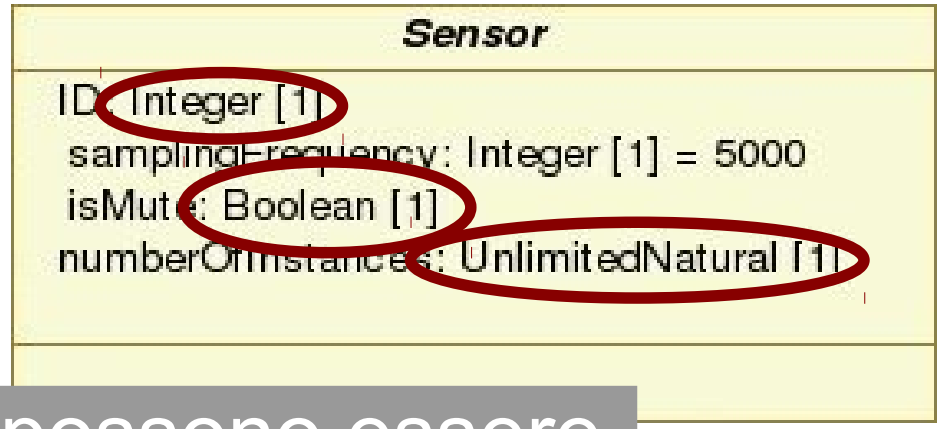
```
public class Sensor {
  Integer ID;
  Integer samplingFrequency = 5000;
  Boolean isMute;
  Integer numberOfInstrances;
}
```



in questo map, l'implementazione Java ricorre ad uno stile O.O. “*puro*”, utilizzando classi per rappresentare i tipi degli attributi e non *tipi primitivi*

comparto attributi – UML && Java

```
public class Sensor {
  int ID;
  int samplingFrequency = 5000;
  boolean isMute;
  int numberOfInstrances;
}
```



Le scelte di interpretazione possono essere delegate all'analista, al progettista, o indotte dagli ambienti di sviluppo utilizzati

comparto operazione

- le operazioni manipolano lo stato degli oggetti (ovvero il valore degli attributi)
  - hanno una segnatura: un tipo, nome e lista di parametri
- operations compartment specifica cosa può fare una classe (e non come), ovvero i servizi che offre
  - vedi slide 11 (operazione, metodo, messaggio)
  - la totale assenza di relazioni per una classe (vedi slide 15)
    - implica che la classe non utilizza altre classi
    - è possibile causa dell'assenza di operazioni che agiscano sullo stato della classe considerata
- ad esempio il nostro sensore :
  - setSamplingFrequency, getMeasure

comparto operazione

- le operazioni manipolano lo stato degli oggetti (ovvero il valore degli attributi)
  - hanno una segnatura: un tipo, nome e lista di parametri

NB : in Java (differentemente da altri linguaggi O.O.) il tipo di ritorno *NON APPARTIENE* alla segnatura del metodo.

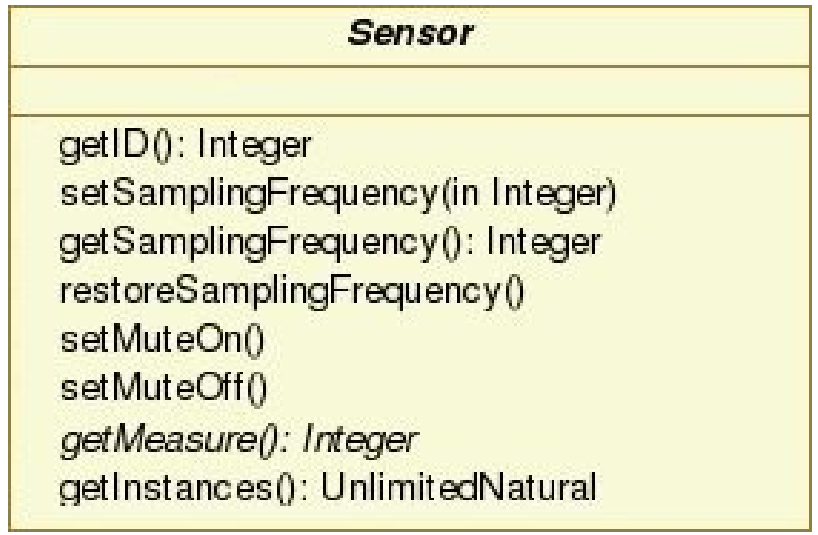
Quindi, ad esempio, non è possibile dichiarare una classe Java con entrambi i metodi:

```
float add(int, int)
int   add(int, int)
```

poiché essi risulterebbero non distinguibili

comparto operazione – UML && Java

```
public class Sensor {
   int getID(){}
   void setSamplingFrequency(int i){}
   int getSamplingFrequency(){}
   void setMuteOn (){}
   void setMuteOff (){}
   int getMeasure(){}
   int getInstances (){}
}
```



costruttori

- sono operazioni speciali
- servono a creare nuove istanze delle classi
  - hanno un ambito di classe e non di istanza (vedi prossima slide 39)
  - pre-esistono alla istanza e non sono “alterabili” dalle classi discendenti
- utili per
  - inizializzare lo stato delle nuove istanze
  - definire un contesto di esecuzione
- non esiste uno standard univoco per la definizione dei costruttori di classe; generalmente :
  - una classe può avere più costruttori
  - hanno tutti lo stesso nome
  - si distinguono per la loro “*segnatura*” (i.e. numero ed ordine dei parametri)
- generalmente rappresentano comportamenti puramente implementativi
  - non vengono esplicitati nei class diagram

costruttori – Java

```
public class Sensor {
  … …
  public Sensor(){
   … …
  }
  public Sensor(int samplingFrequency){
   … …
  }
  … …
}
```

distruttori

- la distruzione di oggetti è più delicata del processo di costruzione
  - problema della consistenza dell'ambiente di esecuzione
- esistono molteplici semantiche differenti per la distruzione di oggetti
- un aspetto comune è che gli oggetti non possano distruggere se stessi ma che la distruzione è sempre delegata ad altre istanze:
  - ✔ C++ : dichiarazione && distruzione esplicita
  - ✗ Java : dichiarazione && distruzione implicita
    - la gestione è totalmente delegata al garbage collector (vedi lezione 2)
- non sono modellati in UML

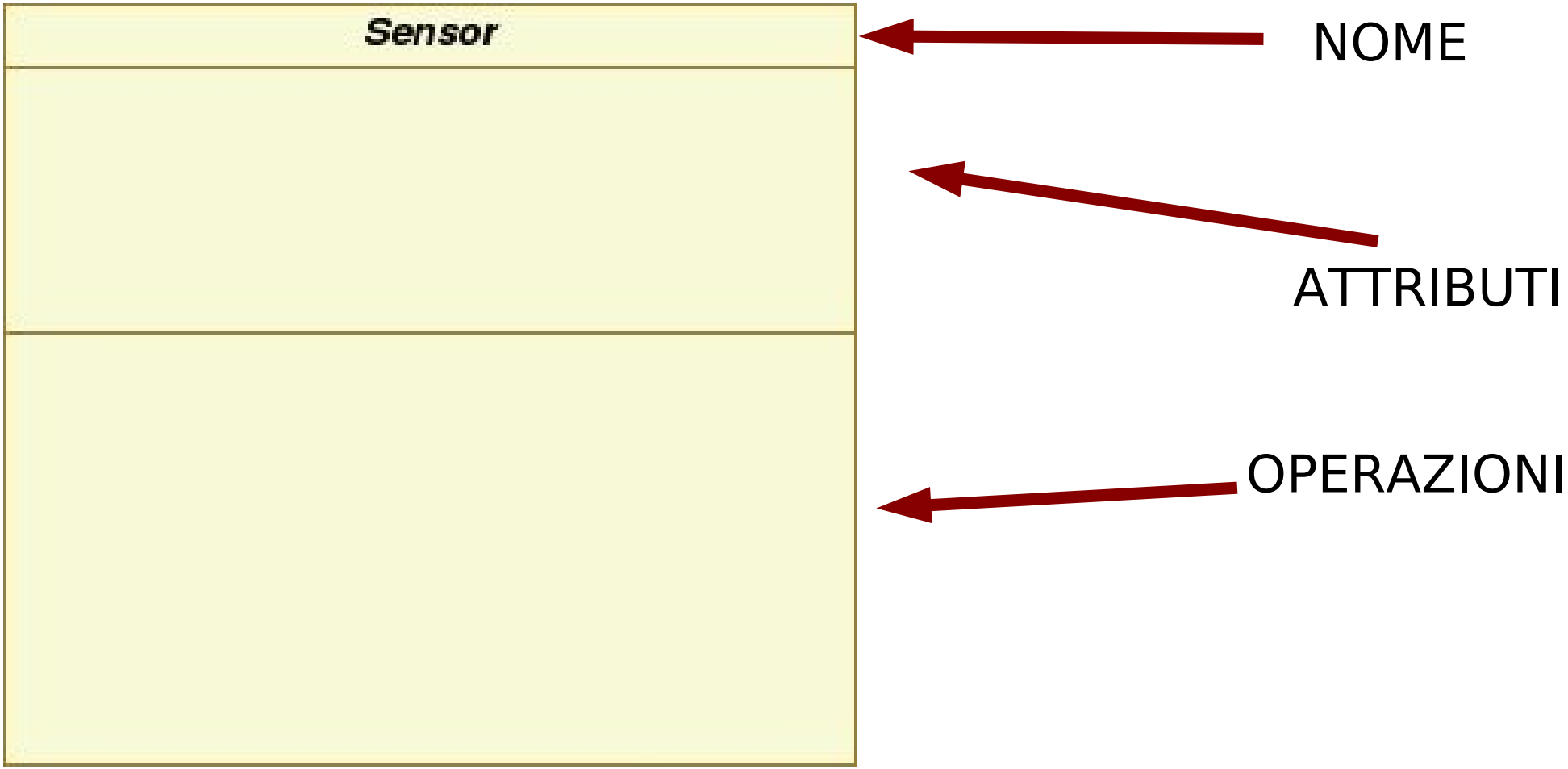
l'istanza speciale `this`

- indica il riferimento all’oggetto stesso
- viene utilizzato nei seguenti ambiti
  - all’interno di un costruttore per invocarne un altro
  - all’interno di metodi e/o costruttori per disambiguare i riferimenti agli attributi ed ai metodi della specifica istanza

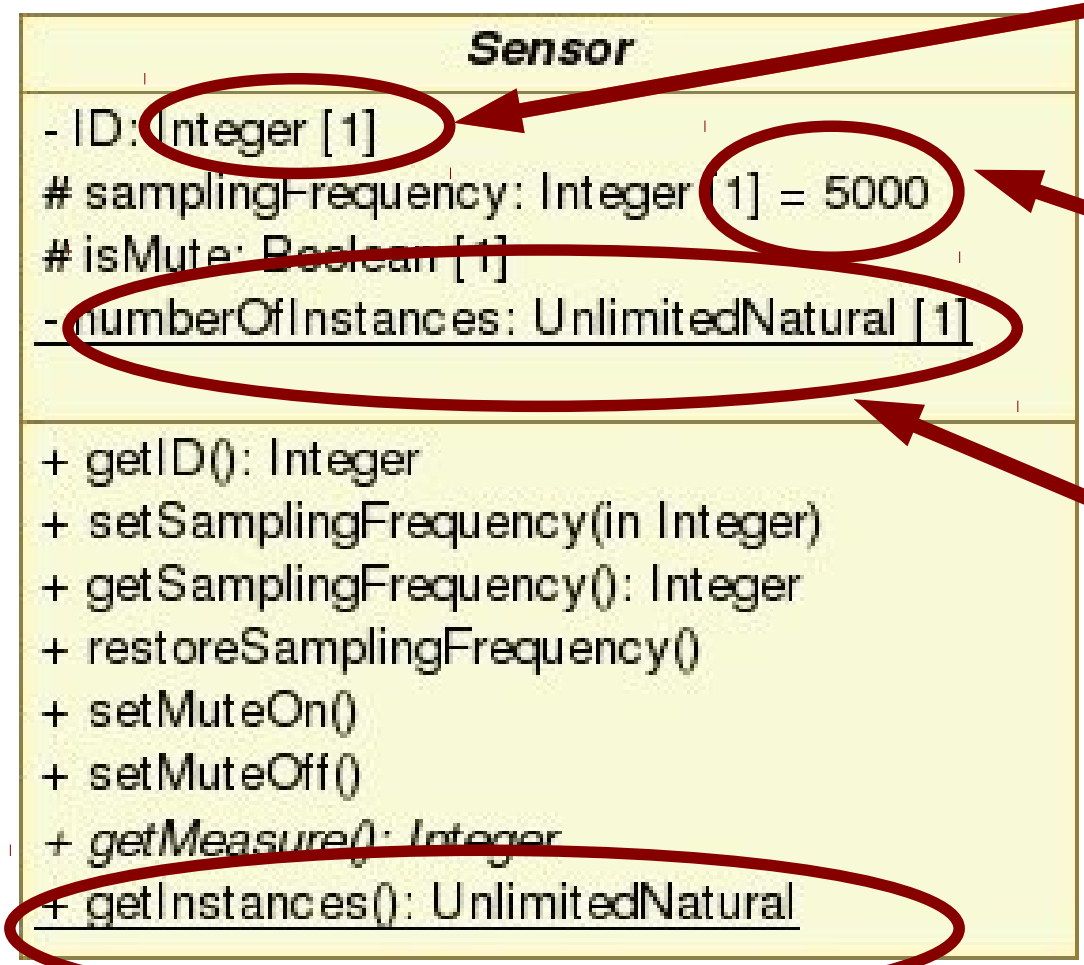
l'istanza speciale `this` - Java

```
public class Sensor {
  int samplingFrequency;
  … …
  public Sensor(){
   this(5000);
   … …
  }
  public Sensor(int samplingFrequency){
   this.samplingFrequency = samplingFrequency;
   … …
  }
  … …
}
```

struttura di una classe – UML (1)



struttura di una classe – UML (2)



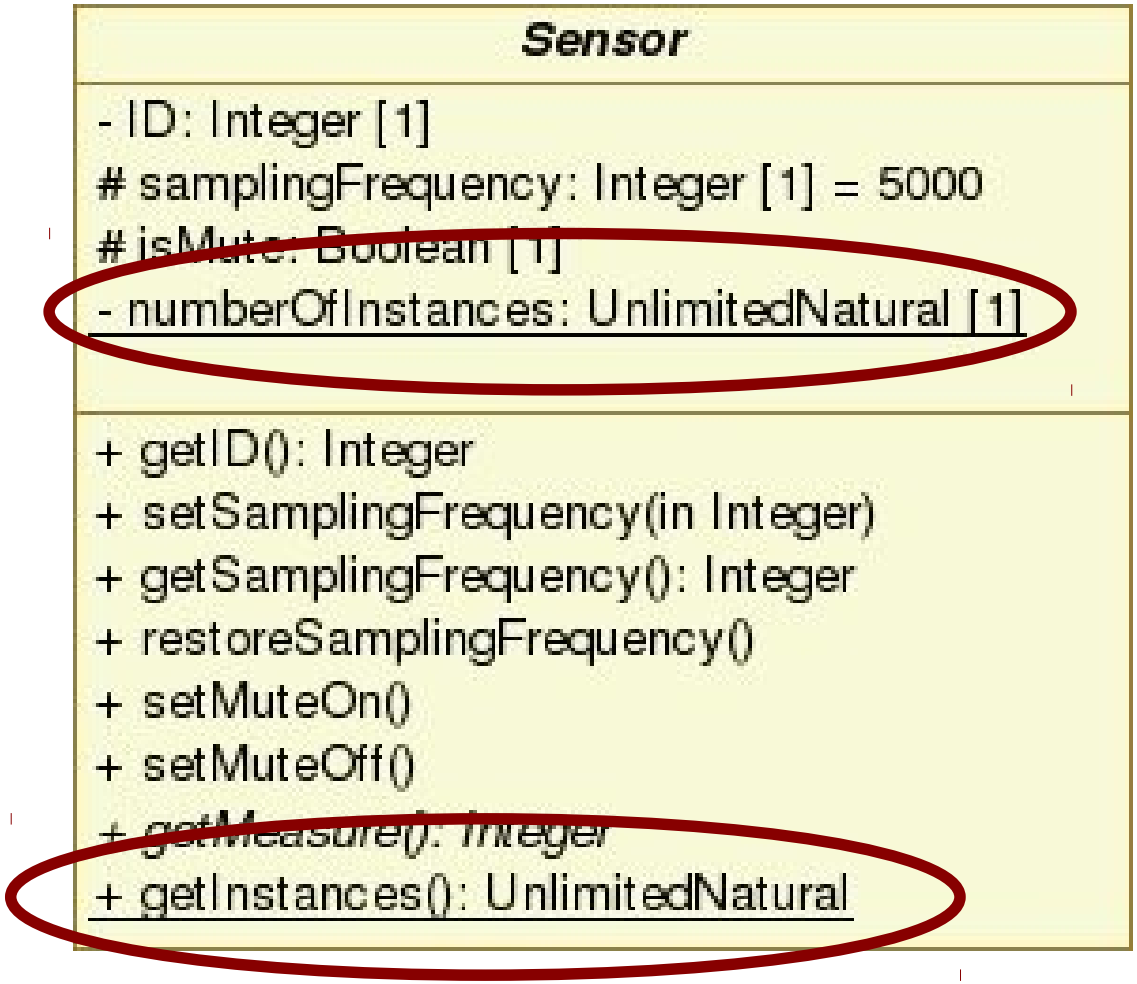
TYPE && MULTIPLICITY

DEFAULT VALUE

ATTRIBUTO DI CLASSE

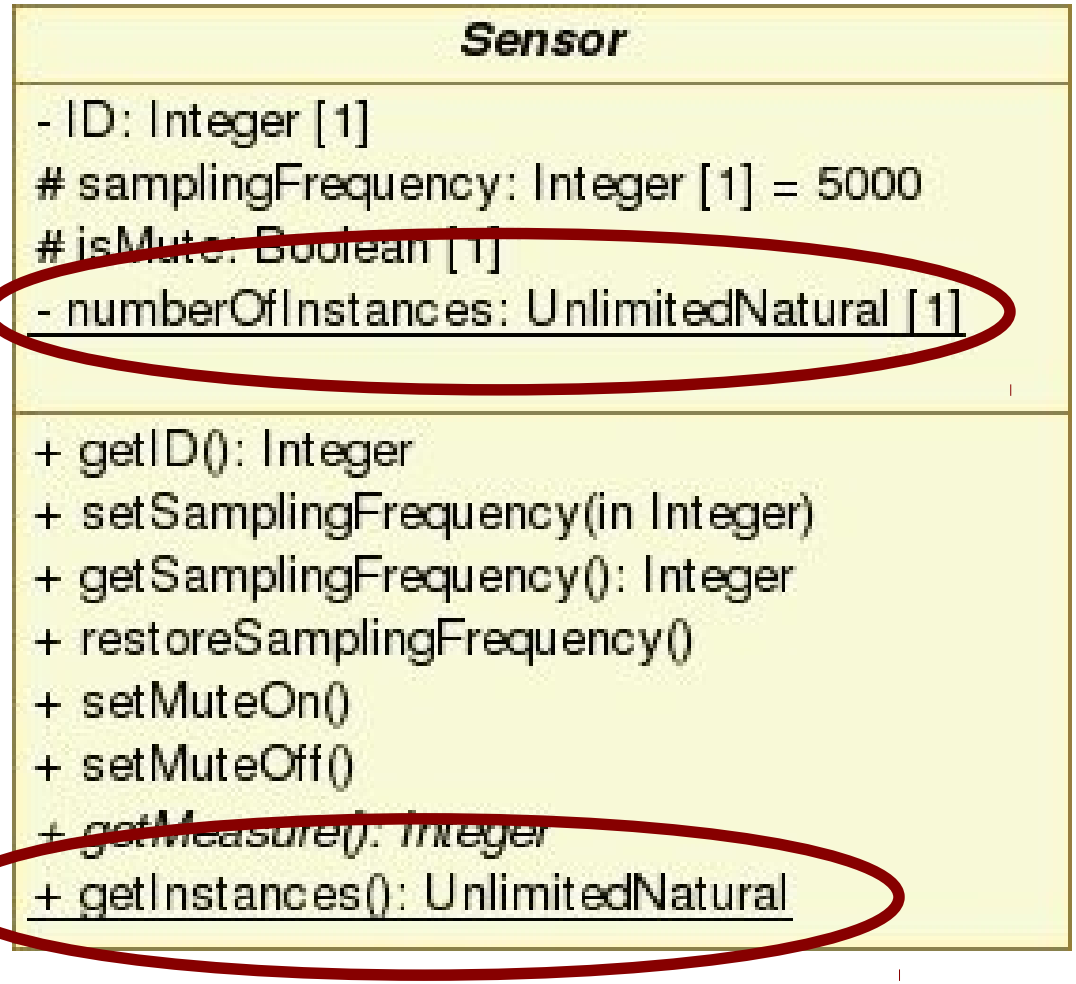
OPERAZIONE DI CLASSE

attributi ed operazioni di classe – UML



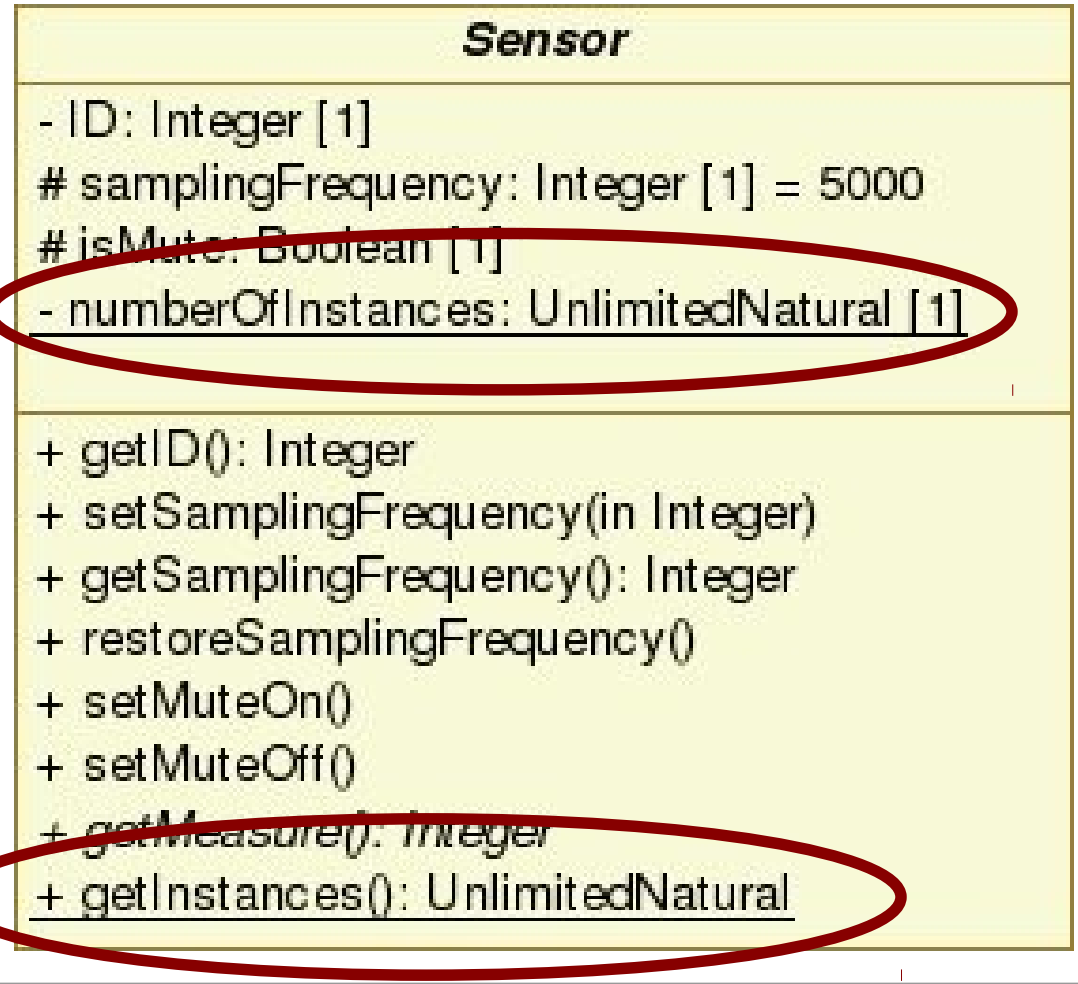
attributi ed operazioni di classe – UML

- attributi : il valore è condiviso tra tutte le eventuali istanze, inoltre non richiedono una istanza della classe per essere impiegati
- operazioni : non richiedono una istanza della classe per essere invocate



attributi ed operazioni di classe – UML

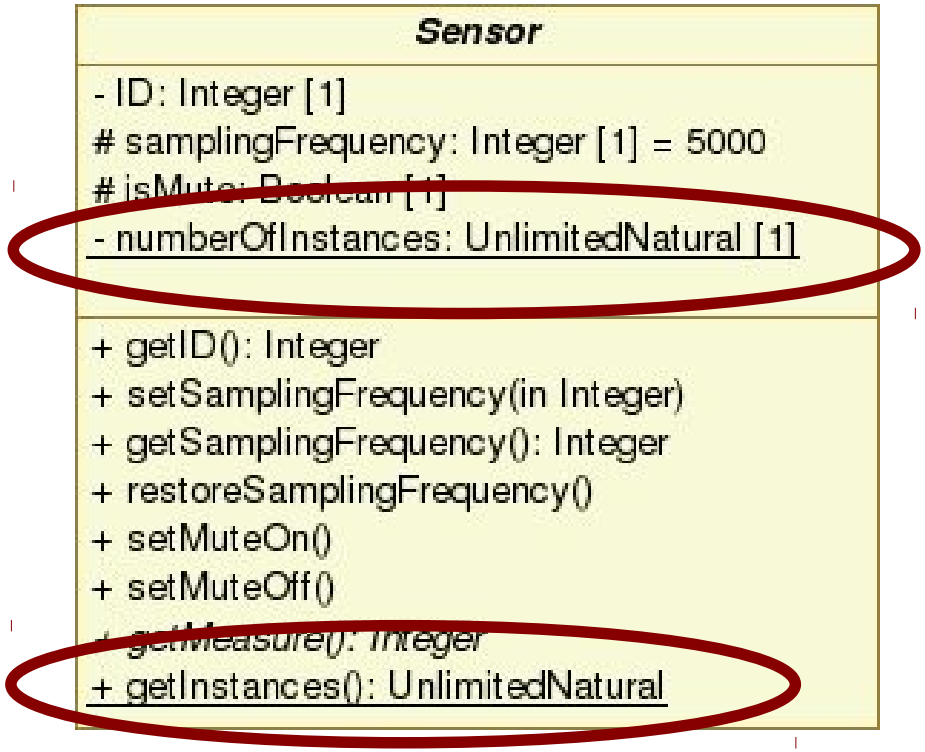
- attributi : il valore è condiviso tra tutte le eventuali istanze, inoltre non richiedono una istanza della classe per essere impiegati
- operazioni : non richiedono una istanza della classe per essere



In pratica, sono "*collegate*" agli invocanti a tempo di compilazione e non a quello di esecuzione.

attributi ed operazioni di classe – map in Java

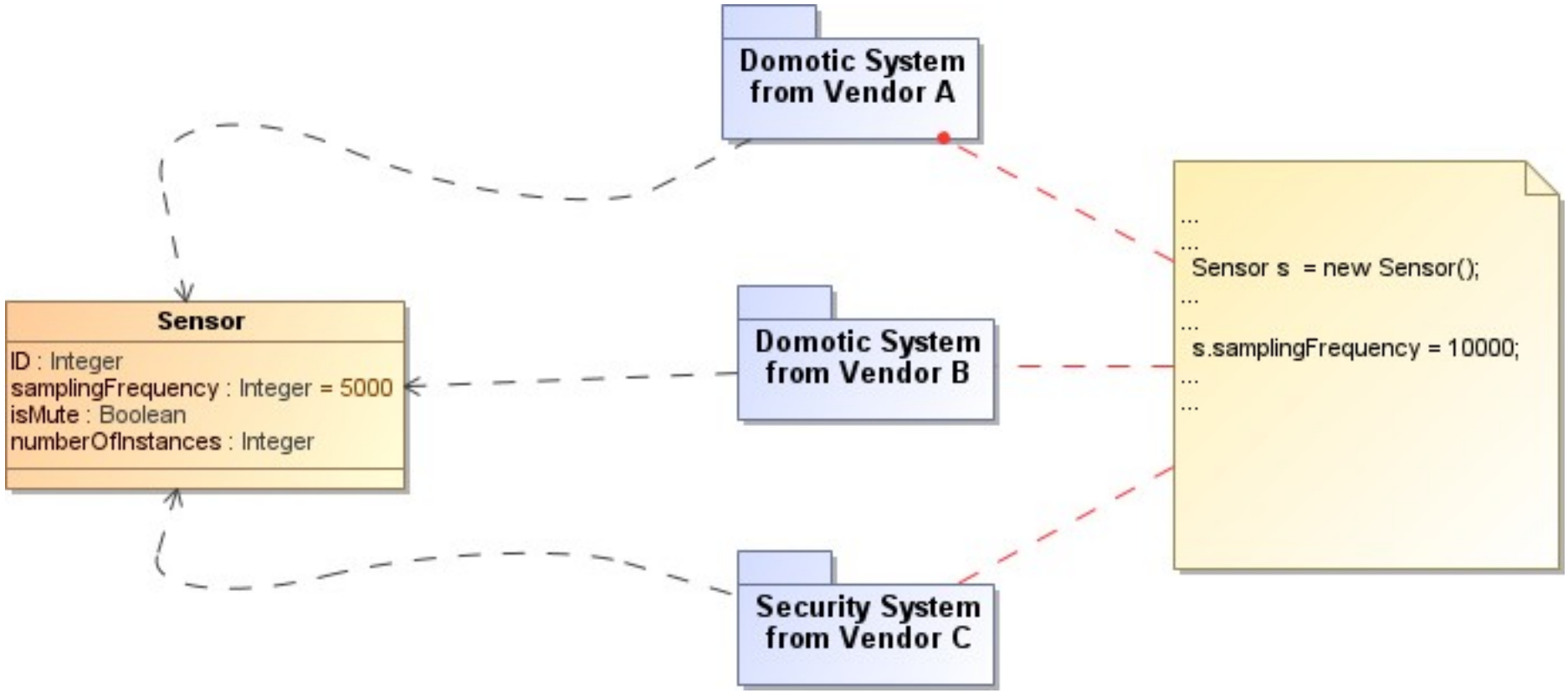
```
public class Sensor {

   ...
   private static int
     numberOfInstances;
   ...
   public static int
     getInstances (){}
}
```



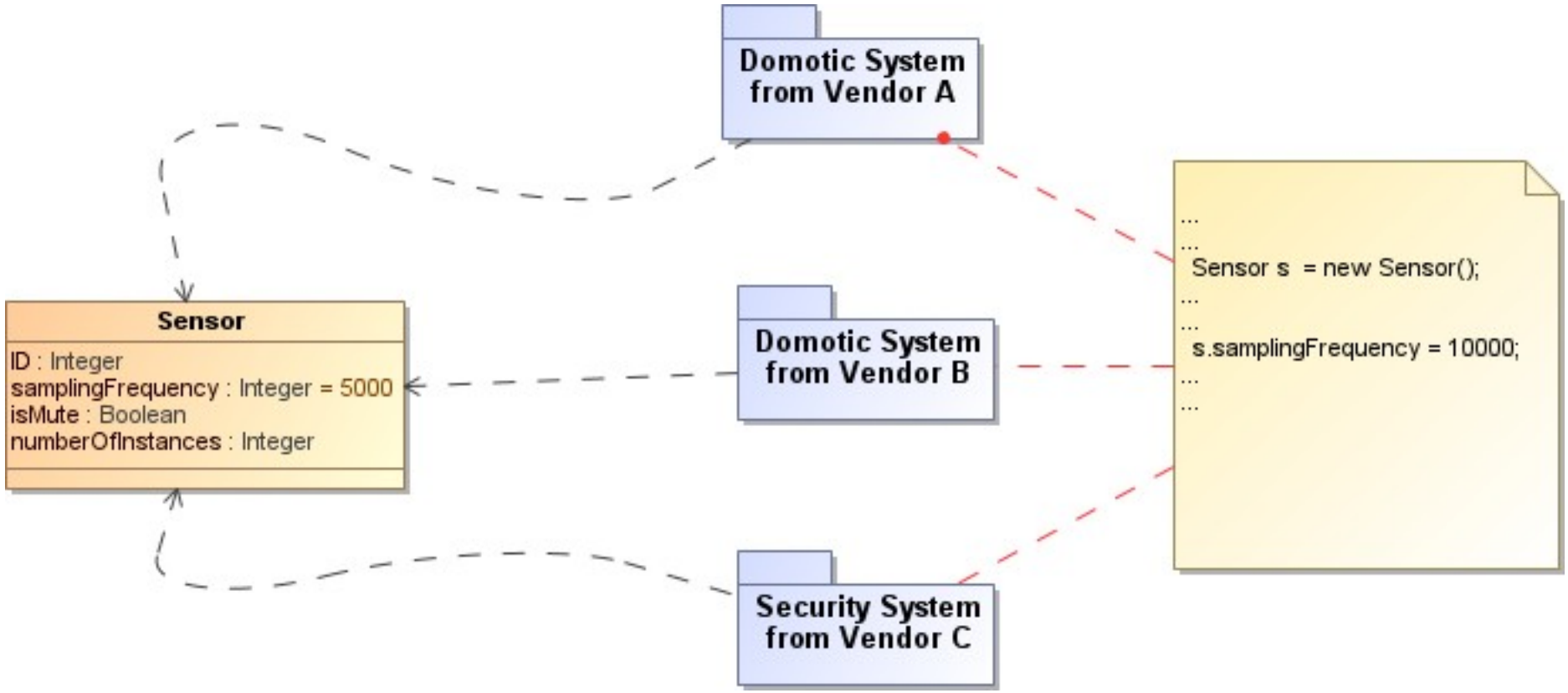
come accederli da un'altra classe

```
int n1 = Sensor.getInstances();
...
Sensor s = new Sensor();
int n2 = s.getInstances();
```

incapsulamento – controllo locale della robustezza

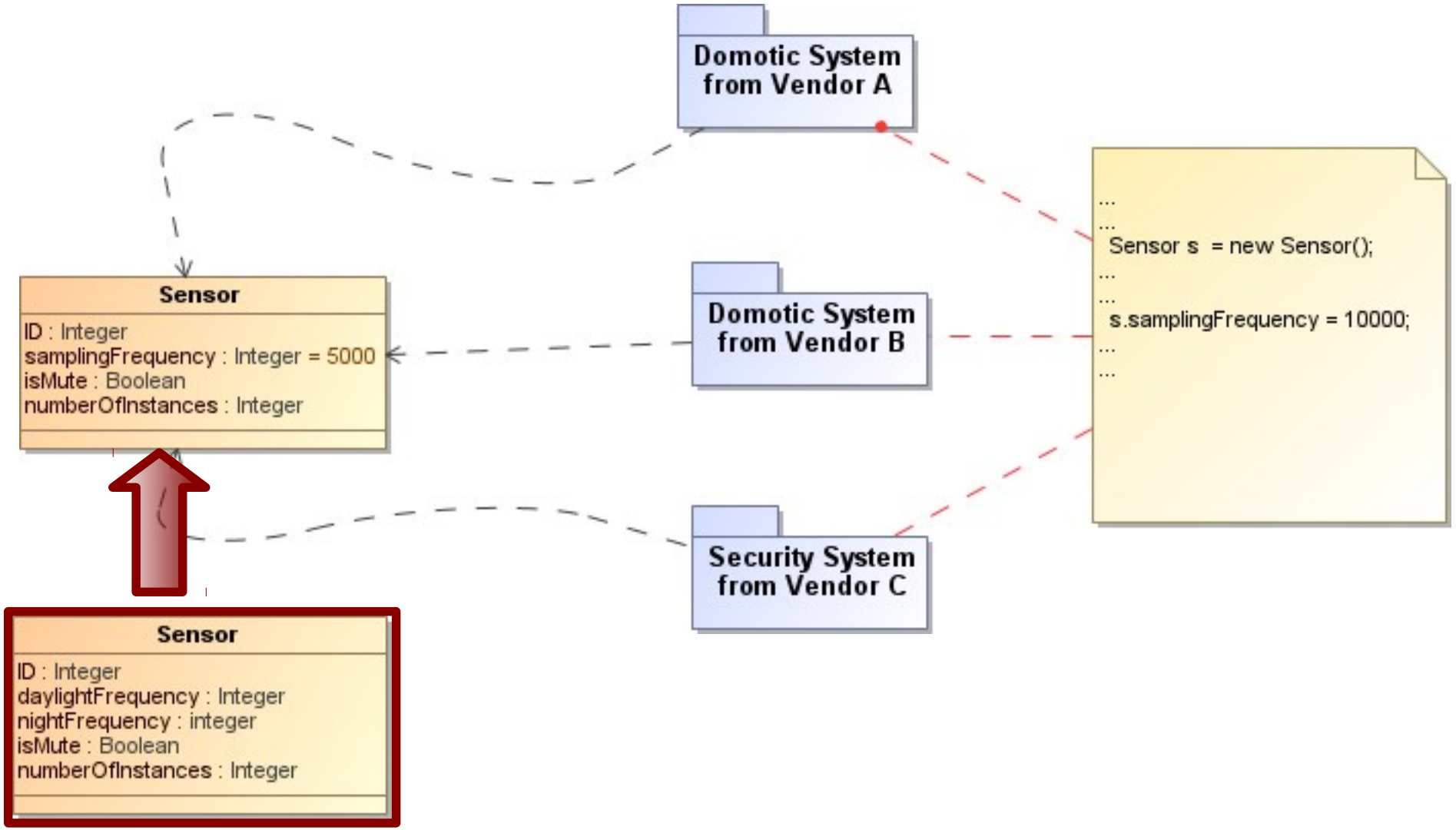


incapsulamento – controllo locale della robustezza

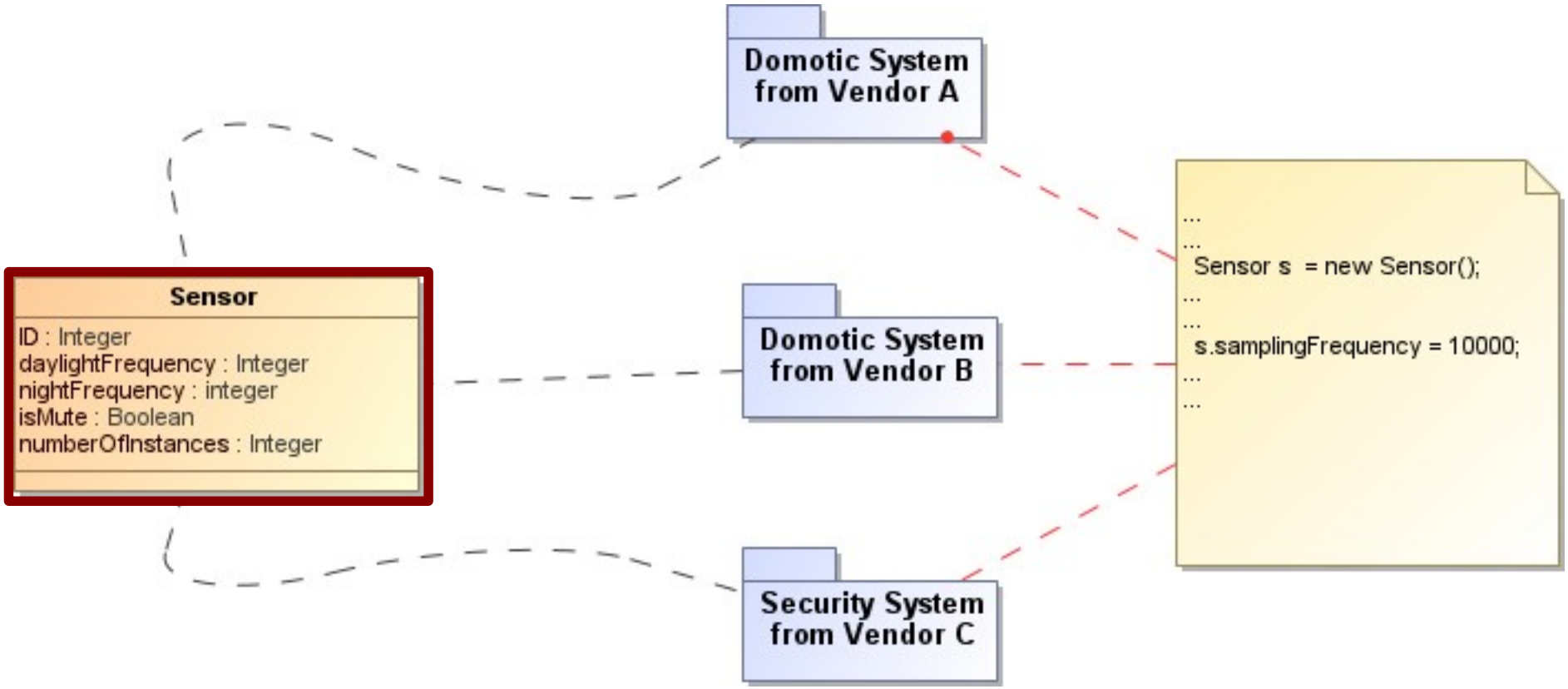


- come gestire e controllare che la corretta interazione con il sensore?
  - e.g. frequenza di rilevamento di non negativa

incapsulamento – località delle modifiche e riusabilità



incapsulamento – località delle modifiche e riusabilità

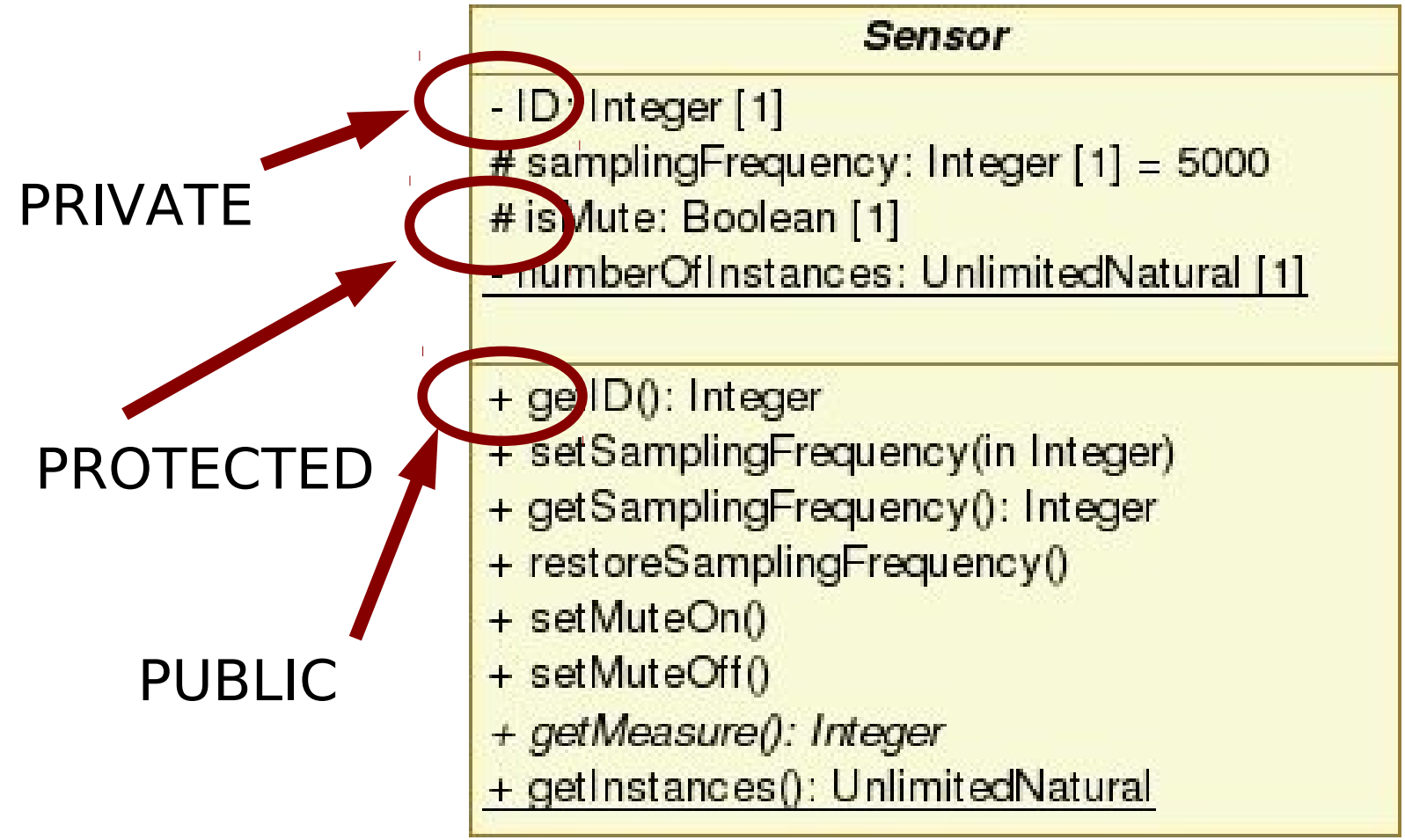


- l'impatto delle modifiche su come Sensor “modella” il suo stato, si estende a tutti i sistemi che “usano” la classe

“vedo” / “non vedo”

- principio di information hiding
  - separazione tra la specifica di una funzionalità e la sua implementazione
  - semantica del dato **VS** implementazione della variabile
- obiettivo : nascondere le scelte che possono essere soggette a cambiamenti

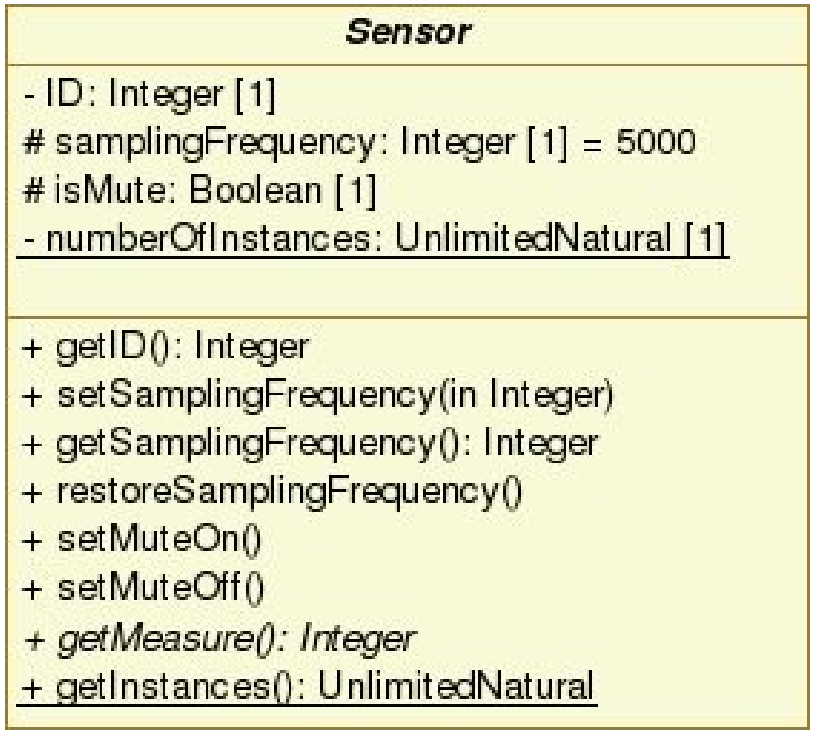
la visibilità di un elemento della classe



visibilità – public && private

(map in Java)

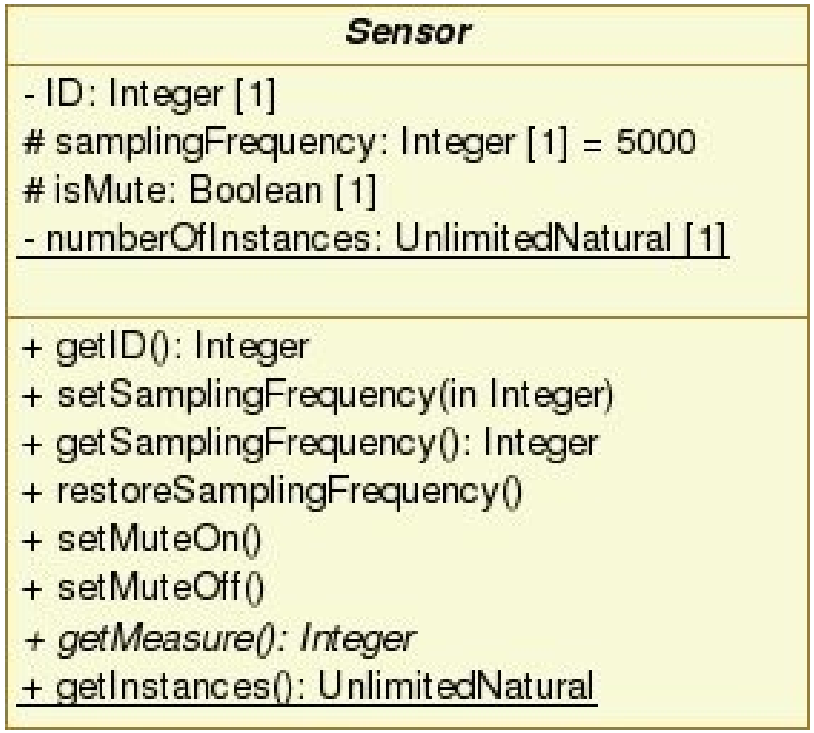
```
public class Sensor {
  public Sensor(){
    this.numberOfInstances ++;
     // OK
  }
}
public class TestClass {
  public void testMethod(){
    Sensor s = new Sensor();
    s.setSamplingFrequency(23);// OK
    s.ID = 45; // ERRORE
  }
}
```



visibilità – protected

(map in Java)

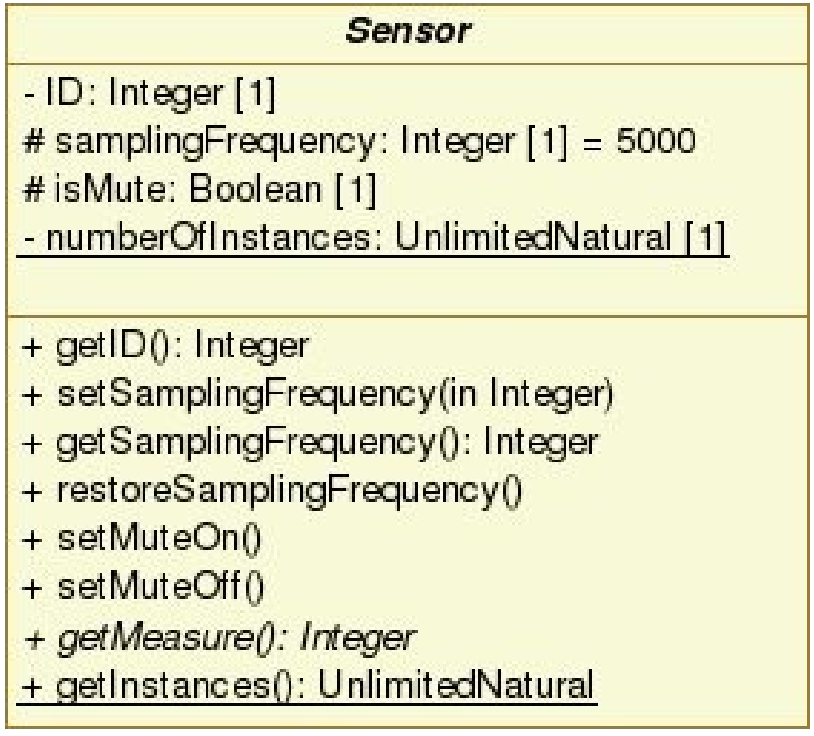
```
public class Sensor {
   public void setMuteOn(){
      this.isMute = true;// OK
   }
}
public class TestClass {
   public void testMethod(){
     Sensor s = new Sensor();
     s.isMute = true ; // ERRORE
   }
}
```



visibilità – protected

(map in Java)

```
public class Sensor {
   public void setMuteOn(){
      this.isMute = true;// OK
   }
}
public class TestClass {
   public void testMethod(){
     Sensor s = new Sensor();
     s.isMute = true ; // ERRORE
   }
}
```



anche se nel caso generale *protected* e *private* sembrano equivalenti, non è cosi. la differenza c'è con la generalizzazione. rimandiamo il discorso alle prossime lezioni.

public, private, protected ?!?!

- principio di information hiding
  - separazione tra la specifica di una funzionalità e la sua implementazione
  - semantica del dato vs. implementazione della variabile
- obiettivo : nascondere le scelte che possono essere soggette a cambiamenti

public, private, protected ?!?!

- principio di information hiding
  - separazione tra la specifica di una funzionalità e la sua implementazione
  - semantica del dato vs. implementazione della variabile
- obiettivo: nascondere le scelte che possono essere soggette a cambiamenti
  - REGOLA PRATICA : gli attributi sono sempre privati, le operazioni possono essere pubbliche
- chiaramente
  - specificare operazioni che enfatizzino il comportamento della classe e non come questo comportamento è implementato
  - l'interfaccia deve essere particolarmente stabile
  - solo se strettamente necessario accedere agli attributi, ma esclusivamente per mezzo di operazioni `get`/`set`

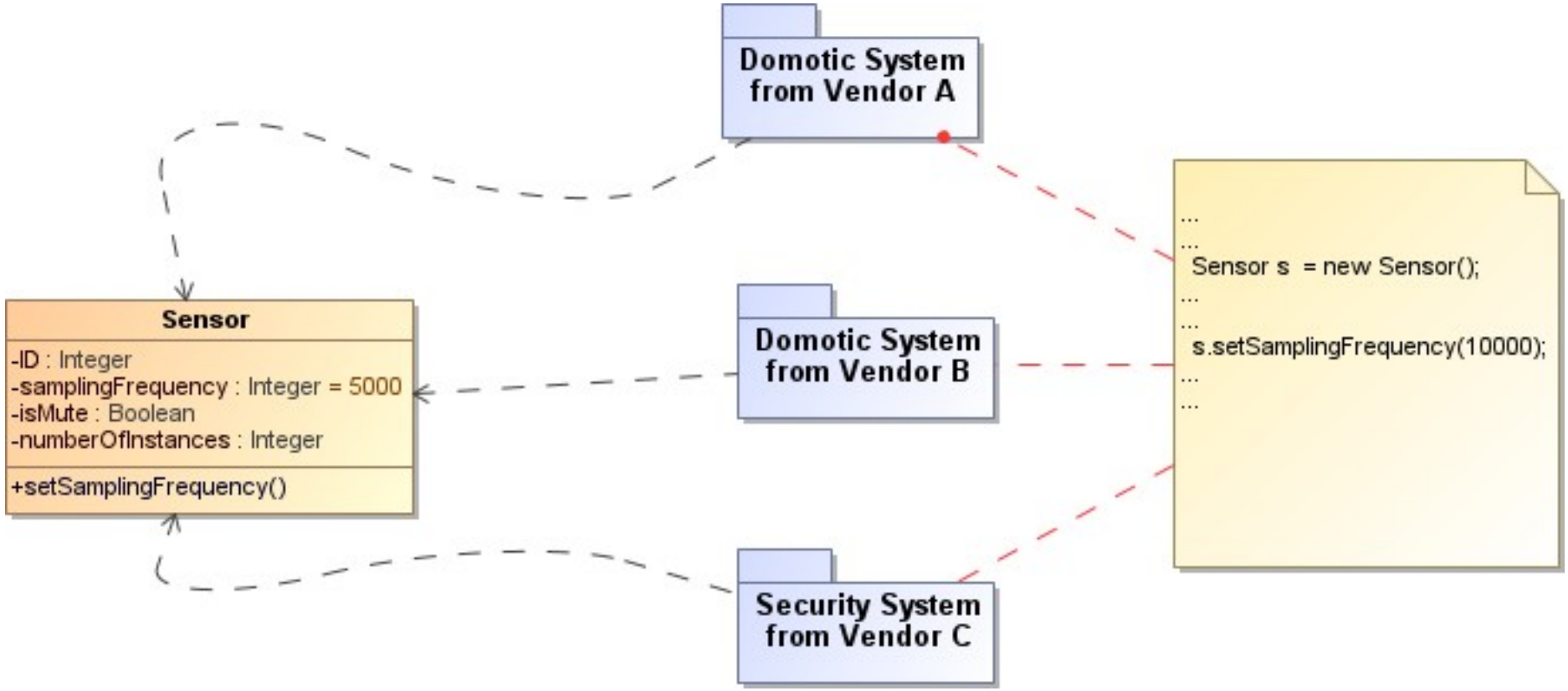
public, private, protected ?!?!

- le operazioni offerte una classe esprimono i possibili modi di interazione con gli altri elementi del sistema
- le operazioni devono essere scelte
  - ✔ per supportare le responsabilità (funzionali e comportamentali) della classe considerata all'interno del sistema
  - ✗ non nella sola ottica di accesso al suo stato (i.e. insieme di attributi privati)
- l'accesso INDIRETTO agli attributi è tipico di particolari classi ausiliarie (i.e. Java Beans, POJO) poste sui confini dell'applicazione
  - sono normalmente utilizzate per gestire aspetti implementativi più che di analisi/progettazione
  - ad esempio per allocarvi COPIE degli attributi di un oggetto per favorire l'interazione (i.e. scambio dati in rappresentazione esterna) con gli attori

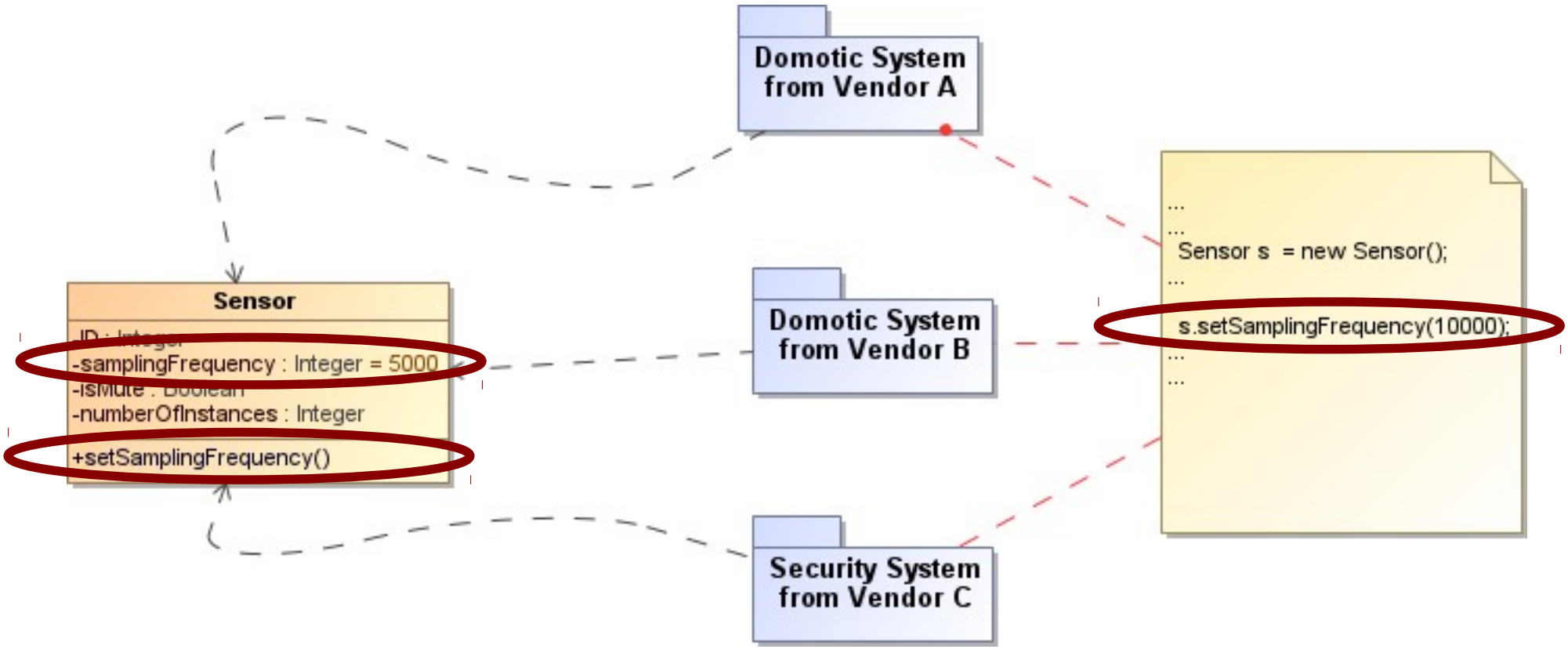
classe e non come questo comportamento è implementato

- l'interfaccia deve essere particolarmente stabile
- solo se strettamente necessario accedere agli attributi, ma esclusivamente per mezzo di operazioni `get`/`set`

incapsulamento – tornando all'esempio



incapsulamento – tornando all'esempio



incapsulamento – tornando all'esempio