Variabili riferimento

- Una variabile riferimento è una variabile il cui tipo è un tipo riferimento.
  - Tipico esempio: oggetto appartenente a una classe
- Una variabile riferimento non contiene memoria dell'oggetto ma solo del riferimento ad esso.
- In pratica, viene allocata un'area di memoria per l'oggetto e la variabile contiene l'indirizzo di questa area di memoria.

Variabili riferimento

- Una variabile riferimento si usa in tre contesti:
  - Memorizzare il riferimento di un nuovo oggetto

```
String s1 = "Hello!";
```

  - Indicare l'oggetto destinatario di un messaggio

```
int n1 = s1.length();
```

  - Come argomento di un metodo o costruttore

```
System.out.println(s1);
```

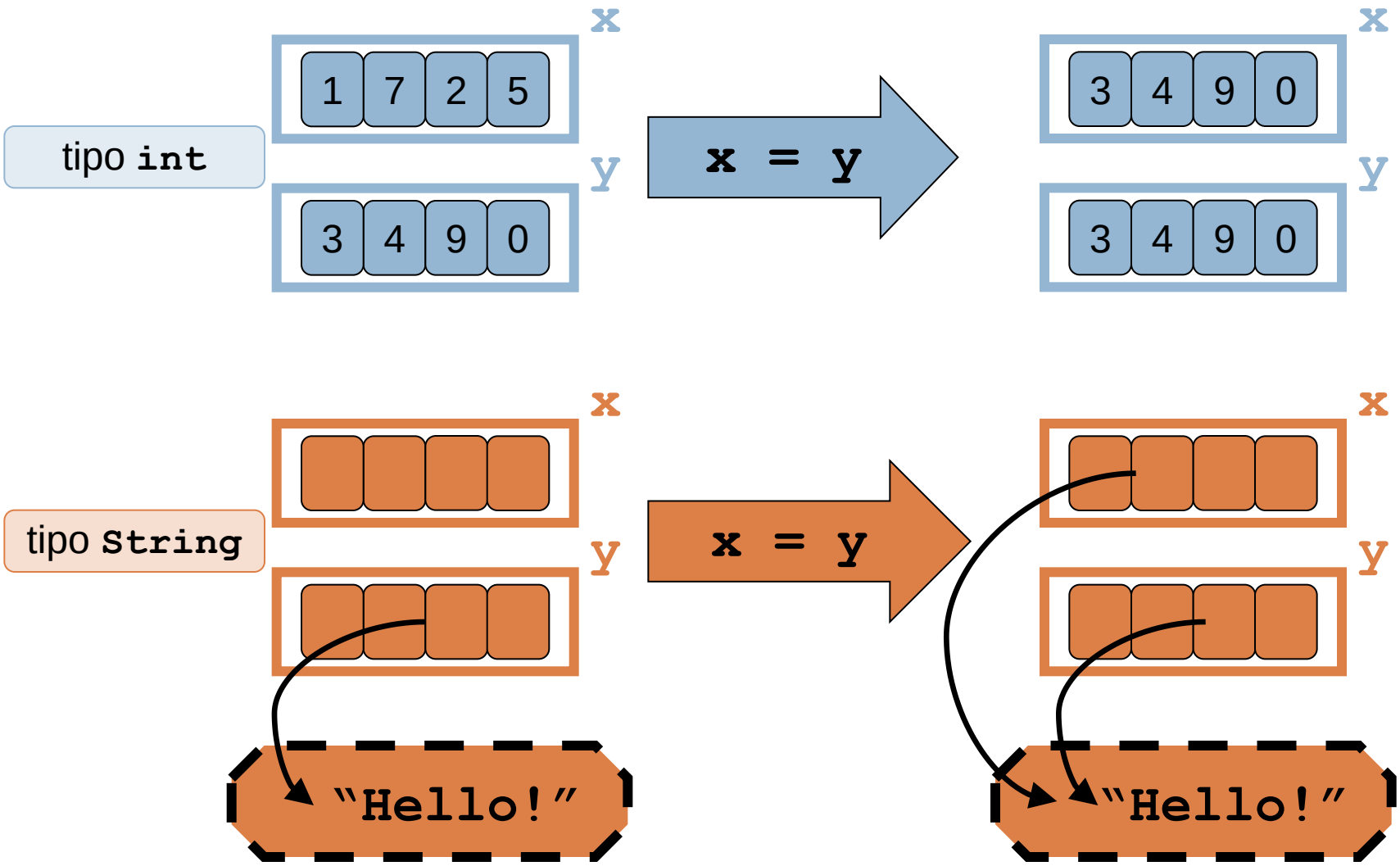
- Si veda ad esempio:

```
String s1 = "Valore = " + s2.compareTo(s3);
```

Variabili riferimento

- L'assegnazione di una variabile riferimento copia il riferimento, non l'oggetto.
- Si consideri l'espressione: `x = y;`
  - Supponiamo che **`x`** e **`y`** siano dello stesso tipo
  - Se **`x`** e **`y`** sono di tipo primitivo, si pone in **`x`** il valore che si trova in **`y`**.
  - Se invece **`x`** e **`y`** sono di tipo riferimento, viene copiato il riferimento.

Assegnamento tipo riferimento



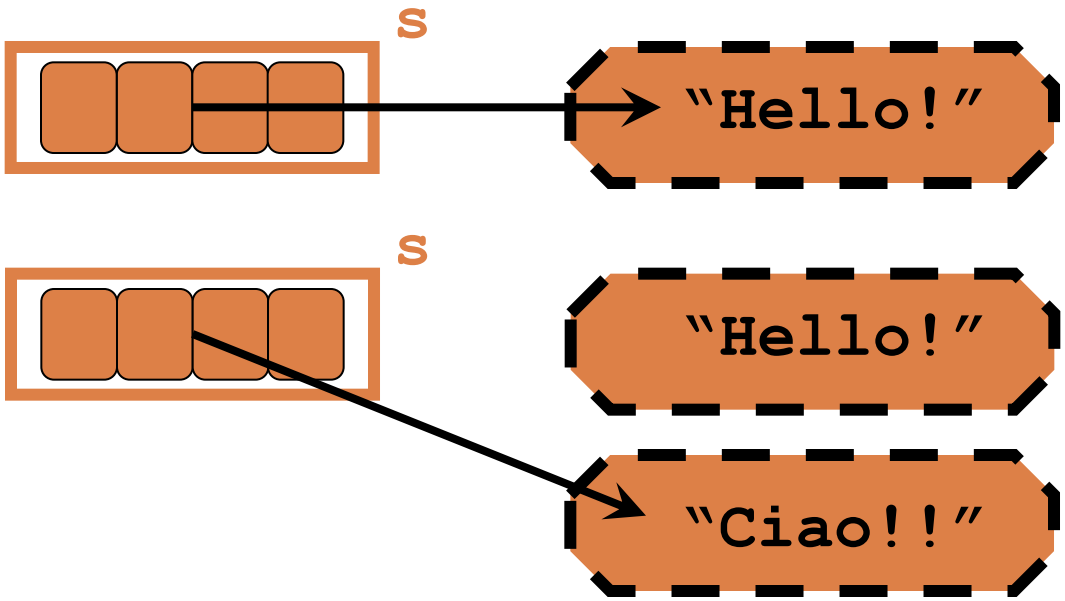
Assegnamento tipo riferimento

```
int x = 5;
x = 18;
```

il valore originario di **x** (cioè 5) viene sovrascritto dal nuovo valore, 18

```
String s = "Hello!";
s = "Ciao!!";
```

viene creato un nuovo oggetto **String**; non viene sovrascritto il valore, ma solo il riferimento

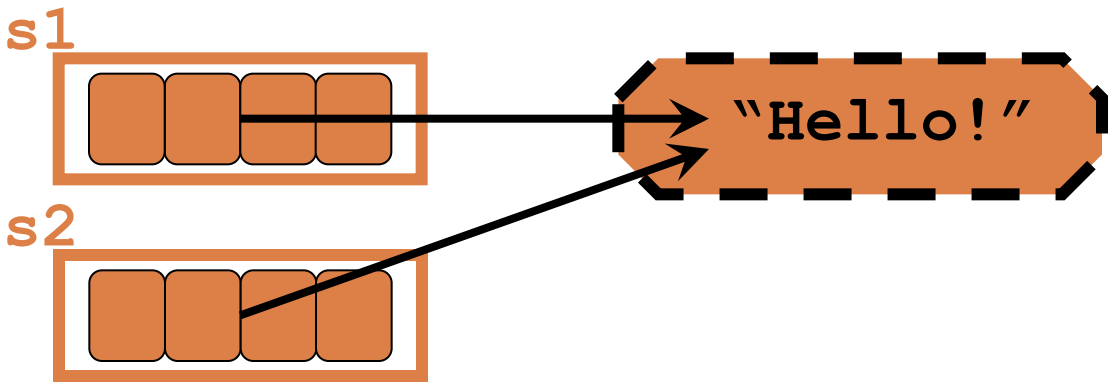


l'oggetto stringa non più referenziato sarà deallocato dal *garbage collector*

Assegnamento tipo riferimento

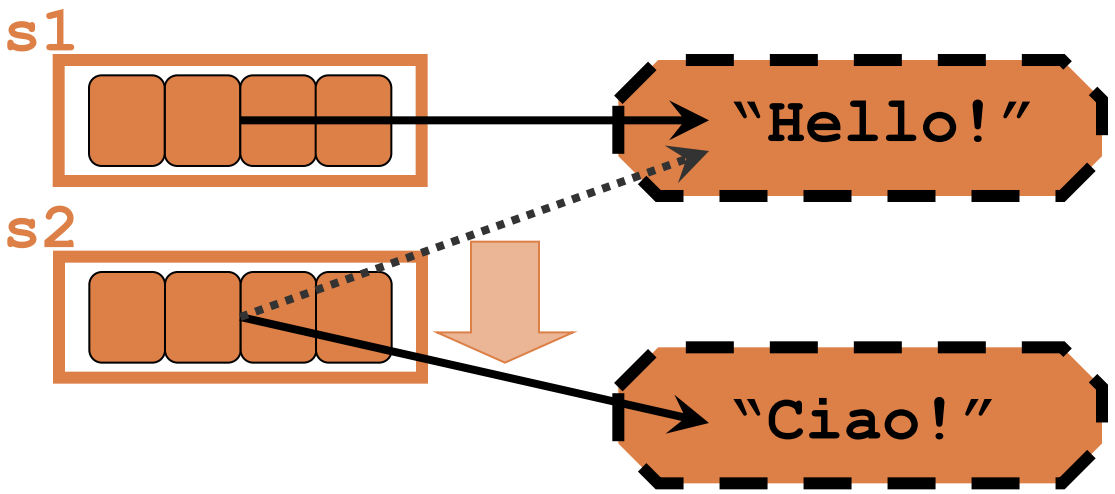
- Le variabili di tipo riferimento sono indipendenti. Un'assegnazione modifica solo il valore di quella variabile, ma non quello di altre variabili che eventualmente referenziano lo stesso oggetto.

```
String s1 = "Hello!";
String s2 = s1;
```



Assegnamento tipo riferimento

```
String s1 = "Hello!";
String s2 = s1;
s2 = "Ciao!";
```



il riferimento di **`s1`** resta all'oggetto **`String`** "Hello!" (ancora valido, perché referenziato)

Copiare oggetti

- Per creare una copia di un oggetto, un assegnamento non è un buon metodo (copia semplicemente il riferimento).
- Per creare davvero una copia si usa tipicamente il metodo `clone`, implementato in molte classi di libreria.

```
MyClass objOne = ...;
MyClass objTwo = (MyClass) objOne.clone();
```

Riferimento `null`

- Una variabile riferimento potrebbe anche non riferire alcun oggetto. Si utilizza un valore speciale, indicato dalla parola riservata **`null`**.
  - **`null`** non vale -1: il suo valore non importa!
- Questo valore non rappresenta alcun oggetto, ed è il valore di inizializzazione di default per le variabili riferimento.
- Assegnare il valore **`null`** a un riferimento significa rilasciare l'oggetto corrispondente (interviene il *garbage collector*)

Riferimento `null`

- Se una variabile riferimento è pari a null, l'oggetto referenziato non esiste: non è possibile invocarne un metodo.
  - Se si chiama un oggetto il cui riferimento è pari a **`null`**, il compilatore ritorna un errore (**`NullPointerException`**).
  - Differenza tra oggetto vuoto e oggetto nullo! Il primo è un oggetto effettivamente esistente (ma vuoto), il secondo non esiste per nulla.

Riferimento `null`

- Esempio illuminante: stringa vuota vs. nulla.

```
String s1, s2;
int x;
s1 = “”; // s1 è la stringa vuota
s2 = null; // s2 è null (lo era già prima)
x = s1.length(); // OK.. ritorna 0
x = s2.length(); // NO.. NullPointerException
```

Altre informazioni sui tipi

- In Java non esiste un operatore "**`sizeof`**": le dimensioni sono fisse (ma ignote) e non bisogna allocare memoria per i dati.
- Il tipo **`boolean`** è un tipo predefinito, i valori **`true`** e **`false`** (case-sensitive) sono parole riservate. Secondo la filosofia di Java, non importa conoscere la loro implementazione. Perciò, **`true`** e **`false`** sono valori intrinsecamente diversi da 0 e 1.
  - Non si può fare casting a **`boolean`**!

Altre informazioni sui tipi

- Il tipo **`char`** rappresenta caratteri.
- Viene usato lo standard UNICODE, uno schema di codifica uniforme multi-lingua per un set di caratteri molto più esteso dell'ASCII.
- I valori si scrivono tra apici: ‘ ’
- Caratteri arbitrari possono essere indicati dalla sequenza **`\u`** seguita da 4 cifre esadecimali, tutto tra apici. Ad esempio: **`'\u2122'`** = ™

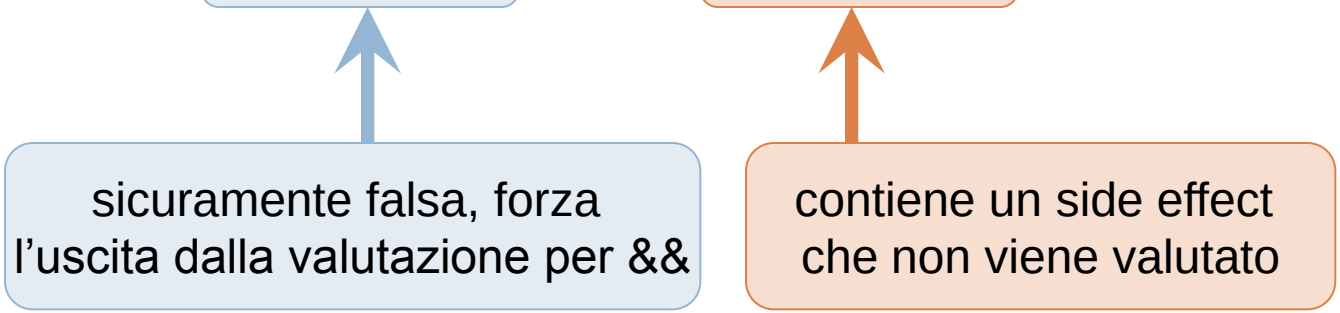
Enunciati condizionali, cicli

- Java implementa tutte le modalità del C per le istruzioni condizionali e i cicli.
- Costrutti **`if`** [**`else`**], **`switch`**, **`?`**: sono identici
  - ricordare che **`else`** chiude sempre l'**`if`** subito precedente;
- Cicli **`for`**, **`while`**, **`do..while`**: identici
  - anche nella possibilità di inserire effetti collaterali
  - dai cicli si può anche uscire con **`break`** e **`continue`**; Java inoltre può usare **`break`** con etichetta (sconsigliati)

Operatori per booleani

- Java utilizza operatori come && e || con significato analogo al C.
- Allo stesso modo, vale la proprietà della **lazy evaluation** (valutazione pigra):

```
if ( (5 < 3) && ( j-- > 0 ) ) ...
```



- Conviene in generale **non avere** side effect.

Operatori per booleani

- Però Java possiede anche operatori & e | che, a differenza del C, sono analoghi a && e ||, solo che forzano l'evoluzione completa dell'espressione (no lazy evaluation).

Progetto di classi

- Che cosa deve rappresentare un oggetto della classe? Quali operazioni deve essere in grado di eseguire?
- La classe deve essere progettata, per quanto possibile, in modo indipendente dalle applicazioni che la utilizzeranno, al fine di garantirne la portabilità.

Progetto di classi

- Interfaccia: cosa è in grado di fare una classe
- Implementazione: come viene realizzata
- L'interfaccia va definita preliminarmente. Chi usa la classe è tenuto a conoscerne l'interfaccia, ma non come viene realizzata.
- La classe quindi deve essere come una **scatola nera** (black box), di cui dall'esterno sono visibili solo **prototipi dei metodi** e **costruttori**.

Progetto di classi

- Progettare una classe significa definire:
  - **cosa**: aspetti e semantiche pubblici della classe
  - **come**: implementazione dei metodi supportati
- La parte che deve essere resa nota è solo il “cosa”: definizione di ciò che il progettista assicura che la classe fa.
- Questo rappresenta il contratto della classe tra il progettista e l’utilizzatore.

Progetto di classi

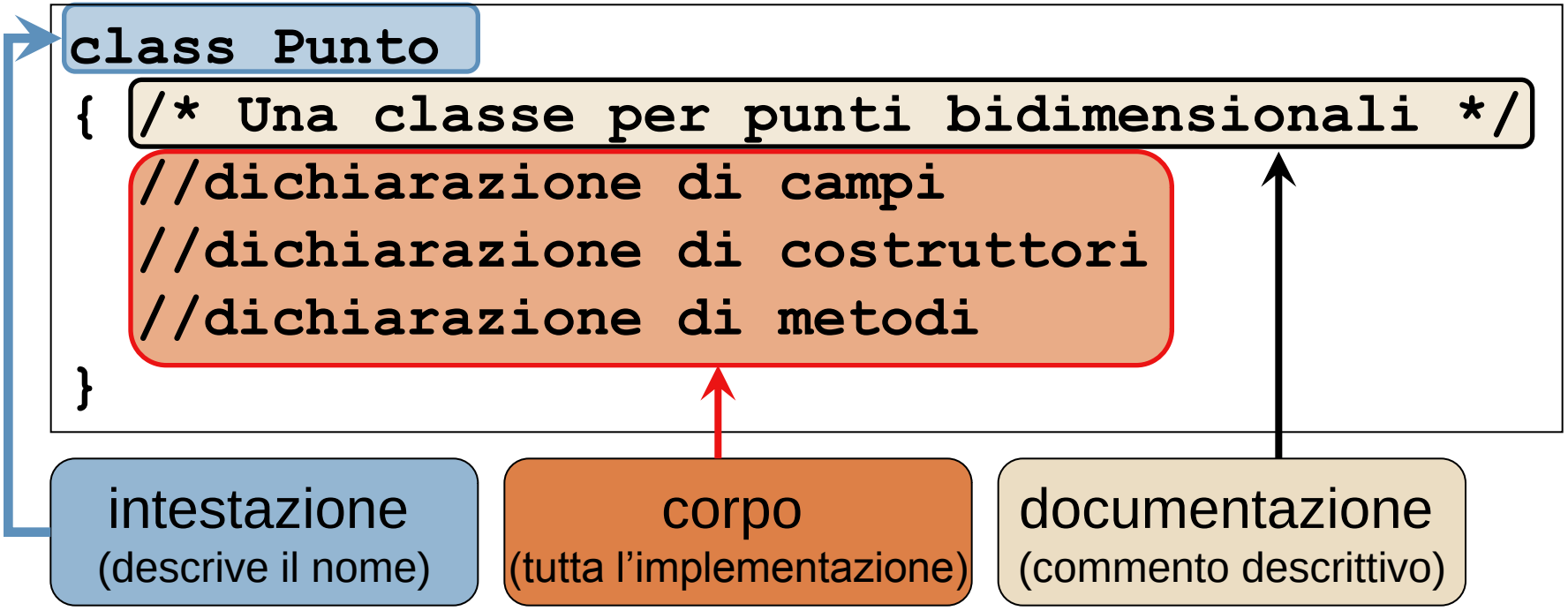
- **Coerenza**: una classe dovrebbe riguardare una singola entità fisica e operazioni simili
- **Separazione dei compiti**: una singola entità può essere, se opportuno, divisa in più classi tra loro correlate, invece di una classe sola
- **Information hiding**: i dati della classe non dovrebbero essere accessibili direttamente, ma solo **tramite metodi** (primato dei metodi sui dati, interazione tramite metodi)

Progetto di classi

- Accesso ai dati tramite metodi. Esisteranno quindi metodi **get** (che ritornano valori) e metodi **set** (che modificano valori)
- Inizializzazione degli oggetti. Conviene che, alla creazione di un oggetto, i suoi campi siano inizializzati in modo parametrico con l'uso di costruttori.

Esempio

- Classe **`Punto`**: definita per applicazioni geometriche, rappresenta punti nel piano cartesiano (bidimensionale).

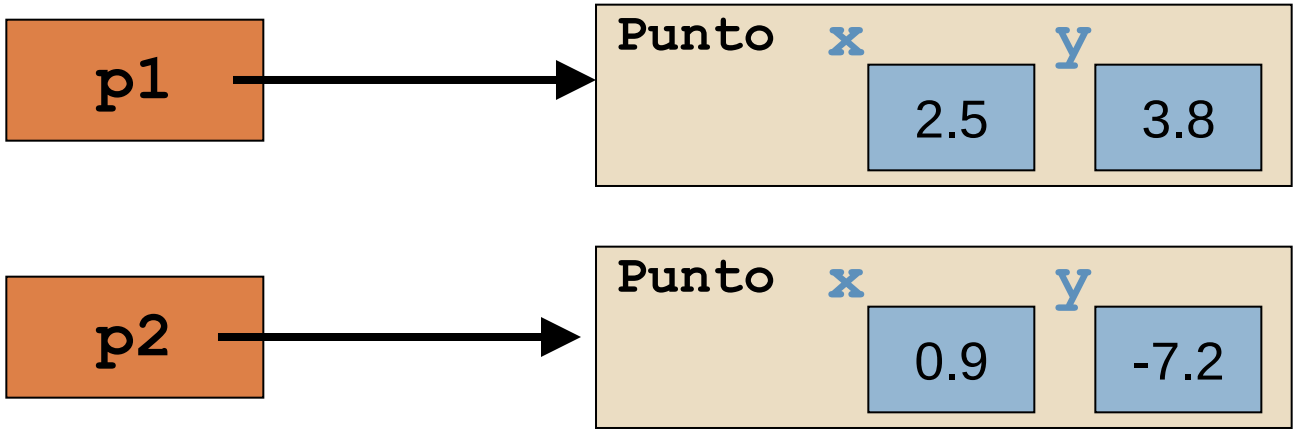


Campi e metodi

- Il dettaglio della progettazione comprende la definizione dello stato (proprietà) e del comportamento (metodi) associati alle istanze della classe.
- Per l'esempio dell'oggetto **`Punto`**:
  - lo stato è una coppia di coordinate reali (x,y) che rappresentano la posizione nel piano
  - il comportamento deve comprendere operazioni per accedere le coordinate e ad esempio operazioni per traslare il punto nel piano

Campi e metodi

- Tutti gli oggetti **`Punto`** hanno le stesse proprietà (ma i valori dei loro stati sono indipendenti tra di loro).



...e sono capaci di eseguire le stesse operazioni.

Campi e metodi

- L'oggetto contiene in memoria solo informazioni sul valore assunto dai campi.
- I metodi sono memorizzati nella classe (in particolare, serve solo il loro bytecode).
- Quando un oggetto deve eseguire un metodo per cui è stato sollecitato dall'opportuno messaggio, la JVM recupera il codice da eseguire nel file `.class` e lo esegue sulla base dello stato dell'oggetto.

Campi e metodi

- Ad esempio se abbiamo due stringhe **`s1`**, **`s2`**, uguali ad esempio a **`"Hello!!"`** e **`"Hi"`** e invochiamo i metodi **`s1.length()`** e **`s2.length()`**:
  - eseguiamo lo stesso codice, quello del metodo **`length()`** per oggetti della classe **`String`**
  - però otteniamo risultati diversi perché lo stato dei due oggetti è diverso

Campi

- Una classe ha all'interno del suo blocco di istruzioni sia dichiarazioni di **campi** (variabili) che dichiarazioni di **metodi.**
- I campi di una classe sono visibili a tutti i metodi dichiarati per quella classe.
- Si parla di **variabili di istanza** (istanza nel senso di "oggetto istanza della classe) se una variabile rappresenta lo stato di un oggetto della classe, cioè un campo di quell'oggetto.

Variabili

- Le **variabili d'istanza** sono variabili come lo sono le **variabili locali** dei metodi.
- Cambia però lo scope:
  - una **variabile locale** è associata a un metodo ed esiste solo nell'ambito di questo
  - una **variabile d'istanza** è associata a un oggetto ed è condivisa da tutti i metodi d'istanza (cioè i metodi non statici) che vengono eseguiti dall'oggetto.

Variabili

- Dentro a un metodo, un identificativo può quindi essere:
  - una variabile di istanza della classe
  - un parametro del metodo
  - una variabile locale
- Oltre alle differenze di scope dovute alla gerarchia, bisogna considerare l'incapsulamento (che vedremo tra poco).

Variabili

```
class Studente
{ //campi
```

`String nome` | `int matricola` | `Date nascita`

```
  //metodi
  void visualizza(int protocollo)
  {  boolean ct = false;
  ...
  //codice del metodo
  ...
  }
}
```

posso accedere **`nome`** (variabile d'istanza), **`protocollo`** (parametro), **`ct`** (variabile locale)

Campi

- Sintassi per accedere una variabile d'istanza:

  `nome_oggetto` . **`nome_campo`**

  (analogamente ai metodi; usabile ovunque)

- Inoltre risulta possibile riferirsi all'oggetto in esecuzione nel codice della classe usando il riferimento **`this`**. Se **`mario`** è di tipo **`Student`**, **`mario.matricola`** viene referenziata nel codice di un metodo dell'oggetto **`mario`** usando **`this.matricola`**

Campi

- Usare **`this`** è un **riferimento esplicito**.
  Il valore di **`this`** viene modificato automaticamente da Java in modo che in ogni istante referenzi l'istanza dell'oggetto in esecuzione durante la chiamata al metodo.
- Se una referenza è non ambigua, Java consente di utilizzare un **riferimento implicito**, cioè omettendo **`this`**.
- Java ricerca una variabile non qualificata risalendo a ritroso i diversi livelli.

Campi

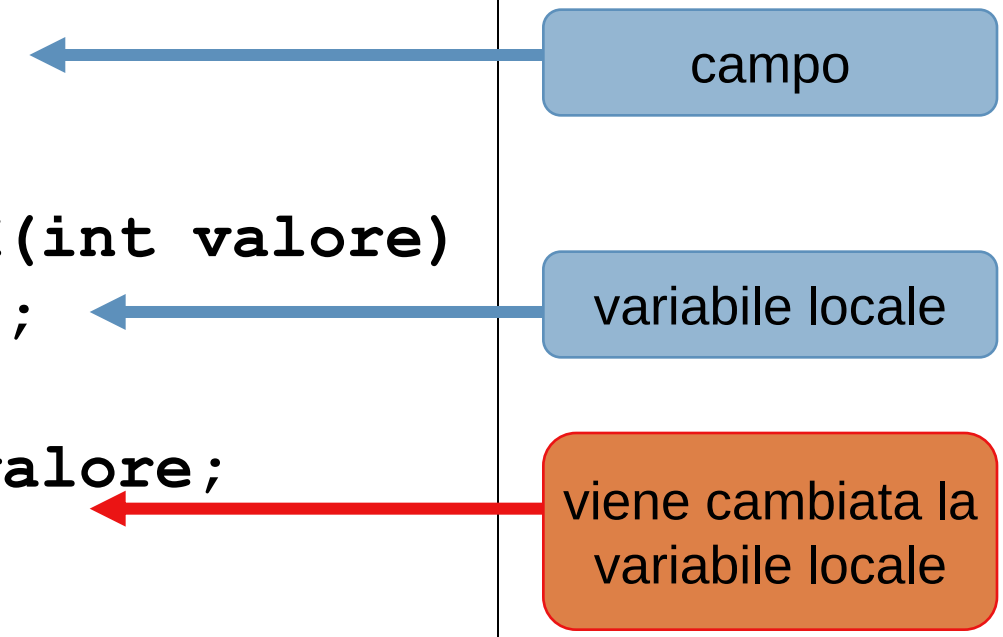
- In pratica, Java cerca una variabile con quell'identificativo tra le variabili locali.
- Se non è presente, risale fino alla lista dei parametri del metodo corrente.
- Se non è presente, Java legge il blocco di dichiarazione dell'oggetto, usando quindi implicitamente **`this`**.
- Se non è presente nemmeno lì, viene generato un codice di errore dal compilatore.

Campi

- Possibile ambiguità (da evitare)

```
class Sbagliata
{
   // campi
   int val;

   //metodi
   void setX(int valore)
   { int val;
      ...
      val = valore;
   }
}
```

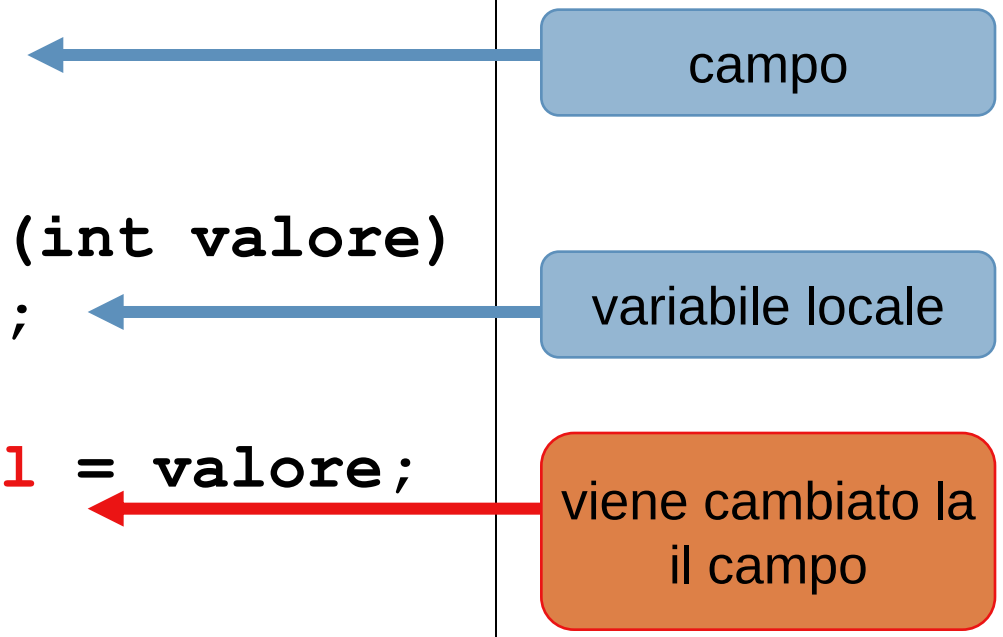


Campi

- Conviene usare un riferimento esplicito:

```
class Giusta
{
   // campi
   int val;

   //metodi
   void setX(int valore)
   { int val;
      ...
      this.val = valore;
   }
}
```



Campi

- I campi vengono automaticamente inizializzati dalla JVM al momento della creazione dell'oggetto.
- Il valore utilizzato dipende dal tipo di campo
  - `0` per i campi di tipo **`int`**
  - `0.0` per tipo **`float`** e **`double`**
  - **`false`** per tipo **`boolean`**
  - `'\u0000'` per tipo **`char`**
  - **`null`** per i campi di tipo riferimento

Campi

```
class MyClass
{
   // campi
   public int x = 1;
   private double y;
   boolean b;
}
```

I campi di **MyClass** sono inizializzati a:

```
x = 1;
y = 0.0;
b = false;
```

- Stiamo inoltre usando le parole chiave **public** e **private** (modificatori di accesso)

Pubblico e privato

```
class Punto
{
 // campi
 public double x,y;
      ...
}
```

```
Punto p1,p2;
p1.x = 0.0;
p1.y = 5.0;
p2.x = p1.x + 4;
System.out.println(p1.x);
```

- Con la definizione di classe **`Punto`** data a sinistra, risulta corretto il codice (inserito in un metodo di un'altra classe) riportato a destra.

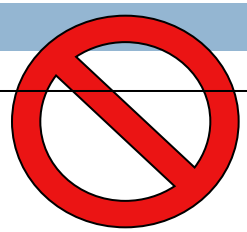
Pubblico e privato

- Però di norma la dichiarazione delle variabili d'istanza viene fatta usando il modificatore di accesso **`private`**, anziché **`public`**.
- In questo caso la variabile può essere acceduta solo da metodi e costruttori definiti nell'ambito della classe stessa.
- L'utente può accedere solo ai **membri pubblici** di una classe (campi, metodi, costruttori dichiarati **`public`**).

Pubblico e privato

```
class Punto
{
 // campi
 private double x,y;
      ...
}
```

```
Punto p1,p2;
p1.x = 0.0;
p1.y = 5.0;
p2.x = p1.x + 4;
System.out.println(p1.x);
```



no modifica!

no accesso!

- Non è più possibile scrivere il codice riportato a destra, perché i campi **`x`** e **`y`** sono dichiarati **`private`**.

Pubblico e privato

- Dichiarando come private le variabili d'istanza di una classe, stiamo imponendo che le informazioni associate agli oggetti istanza della classe non possano essere accedute da tutti (**incapsulamento dell'informazione**).
- In effetti, per manipolare gli oggetti è bene non farlo direttamente accedendo i loro campi (che sono privati) ma utilizzando opportuni metodi pubblici definiti dall'oggetto stesso.

Pubblico e privato

- Principio basilare dell'OOP! Gli oggetti vanno usati tramite la loro interfaccia, ignorando i dettagli della loro implementazione.
- Di conseguenza:
  - è **`public`** ciò che riguarda l'interfaccia
  - è **`private`** ciò che è un dettaglio di implementazione

Pubblico e privato

- L'incapsulamento si realizza utilizzando **metodi pubblici** per leggere e modificare **campi privati**. In particolare si usano:
  - metodi di accesso (**get**) che restituiscono al chiamante informazioni sullo stato di un oggetto; di solito, questa informazione viene fornita nel valore di ritorno del metodo
  - metodi di modifica (**set**) che cambiano lo stato dell'oggetto a cui sono applicati

Pubblico e privato

- Definendo i metodi get e set nel modo opportuno, il progettista ha la possibilità di:
  - decidere come i dati vengono acceduti (privacy: comunico i dati nel modo che ritengo opportuno; alcuni dettagli implementativi posso anche non comunicarli)
  - evitare modifiche inopportune dei dati ed effetti collaterali vari (le modifiche sono solo quelle previste dal metodo **set**)