

Programmazione Orientata agli Oggetti

Oggetti e Riferimenti

Sommario

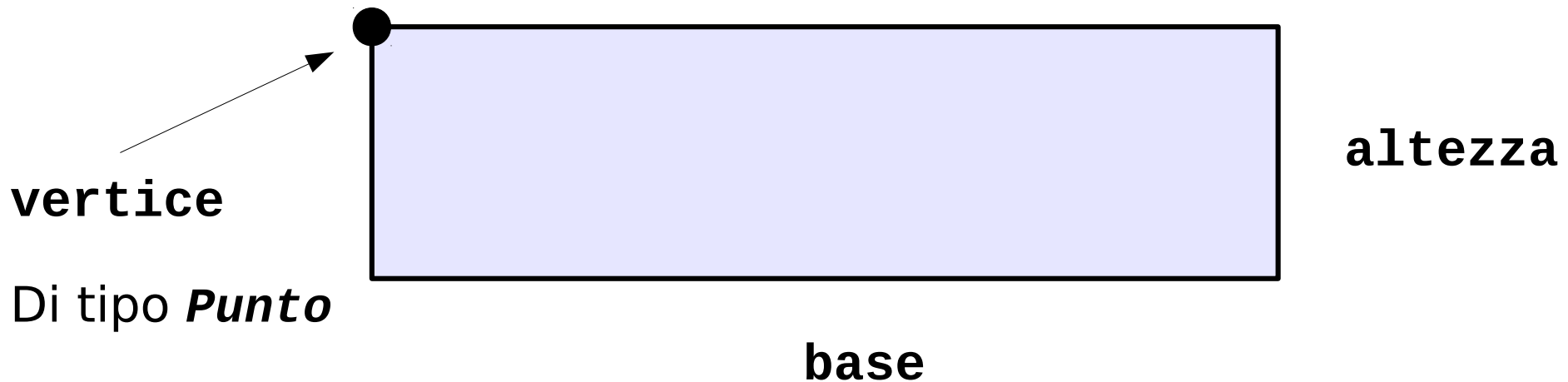
- Gli oggetti in *Rete*
- Stato degli oggetti
 - Variabili di istanza
 - Inizializzazione
 - Campo d'Azione (*Scope*)
- Comportamento degli oggetti
 - Metodi
 - Parametri
 - Variabili Locali
- Riferimenti
 - Riferimenti ad un oggetto, *side-effect*
 - Riferimenti, parametri e valore di ritorno
 - Riferimento **null** e **NullPointerException**
 - La parola chiave **this**

Gli Oggetti in *Rete* (1)

- La *definizione* di un programma “orientato agli oggetti” consiste nella definizione di diverse *classi* di oggetti
- L'*esecuzione* di un programma orientato agli oggetti avviene orchestrando lo scambio di messaggi tra un pluralità di oggetti istanza delle classi definite nel programma
 - gli oggetti devono “conoscersi”
 - un oggetto può possedere *riferimenti* verso gli altri oggetti
 - gli oggetti possono inviare messaggi ad altri oggetti dei quali possiedono un *riferimento*
- Si vuole realizzare la classe **Rettangolo**
 - Stato composito:
 - base
 - altezza
 - posizione del vertice in alto a sx

Gli Oggetti in *Rete* (2)

- Il vertice in alto a sinistra è un oggetto istanza della classe **Punto**
 - di coordinate (x, y)
- Ogni oggetto della classe **Rettangolo** deve conoscere un oggetto della classe **Punto** che rappresenta il suo vertice in alto a sinistra



Gli Oggetti in *Rete* (3)

oggetto



*riferimento
ad oggetto*

oggetto



(Con Eclipse) La Classe Rettangolo

- Nello stesso progetto della classe **Punto**

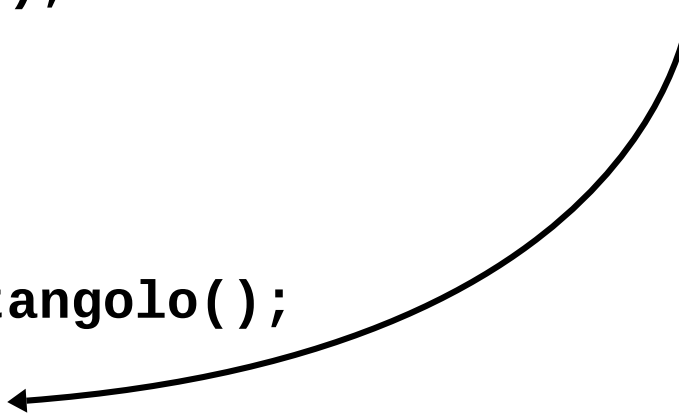
```
public class Rettangolo {  
    private int base;  
    private int altezza;  
    private Punto vertice;  
  
    public void setBase(int b) { base = b; }  
    public void setAltezza(int a) { altezza = a; }  
    public void setVertice(Punto v) { vertice = v; }  
  
    public int getBase() { return base; }  
    public int getAltezza() { return altezza; }  
    public Punto getVertice() { return vertice; }  
}
```

La Classe Rettangolo (2)

- Il `main()` può essere modificato come segue

```
public class MainForme {  
    public static void main(String[] args) {  
        Punto origine = new Punto();  
        origine.setX(0);  
        origine.setY(0);  
  
        Rettangolo rect = new Rettangolo();  
        rect.setVertice(origine);  
        rect.setBase(8);  
        rect.setAltezza(3);  
        // ...  
        // Seguono stampe per verificarne il funzionamento  
    }  
}
```

Dopo questa istruzione l'oggetto istanza di **Rettangolo** conosce l'oggetto istanza di **Punto**



Messaggi tra Oggetti (1)

- Si vuole implementare il metodo **sposta(int deltaX, int deltaY)** nella classe **Rettangolo**
 - Per traslare il suo vertice il rettangolo può chiedere al suo stesso vertice di spostarsi: scambio di messaggi tra oggetti

```
public class Rettangolo {  
    // ... Come prima ...  
    public void sposta(int deltaX, int deltaY) {  
        vertice.trasla(deltaX, deltaY);  
    }  
}
```


Messaggi tra Oggetti (2)

- E se la classe **Punto** non disponesse del metodo **trasla()**?

Messaggi tra Oggetti (3)

```
public class Rettangolo {  
    // ... come prima ...  
    public void sposta(int deltaX, int deltaY) {  
        int xVertice = vertice.getX();  
        int yVertice = vertice.getY();  
        vertice.setX(xVertice + deltaX);  
        vertice.setY(yVertice + deltaY);  
    }  
}
```

- Il comportamento desiderato è comunque ottenibile utilizzando i metodi `setX()` e `setY()` ma il codice risulta “meno pulito” rispetto alla soluzione basata sulla disponibilità del metodo `trasla()` già all’interno della classe **Punto**

Variabili di Istanza e Metodi

- Una classe definisce sia delle variabili di istanza sia dei metodi; rappresentano, rispett.
 - *lo stato* degli oggetti istanza di quella classe
 - *il comportamento* di tali oggetti
- La definizione di una classe segue questa sintassi difatti:

```
public class <NomeClasse> {  
    <definizione di variabili di istanza>  
    <definizione di metodi>  
}
```

Variabili di Istanza (1)

- Le variabili di istanza memorizzano informazioni che rappresentano lo stato di un oggetto

```
public class Rettangolo {  
    private int base;  
    private int altezza;  
    private Punto vertice;  
    ...  
}
```

Variabili di Istanza (2)

- Nella definizione di una variabile di istanza:
 - Modificatore di visibilità (>>)
 - Tipo
 - Nome della variabile

private int base;



Modificatore di visibilità

Tipo

Nome della variabile

- Il modificatore di visibilità specifica se una certa variabile è visibile dall'esterno
 - Per il momento si usa **private**: la variabile è visibile solo all'interno della classe in cui è dichiarata
 - Per accedervi dall'esterno si usano i metodi *getter* e *setter*

Variabili Istanza: Inizializzazione (1)

- Ci sono diversi modi per inizializzare lo stato di un oggetto
- Le variabili di qualsiasi genere (ovvero di istanza, locali ed altro >>) vengono *sempre* inizializzate, esplicitamente od implicitamente
- In Java non esiste il problema delle variabili accidentalmente rimaste non inizializzate tipico del linguaggio C
- Per le variabili di istanza: se non viene specificato alcun valore iniziale, assumono un valore di default:
 - per le variabili di un tipo numerico (**int**, **float**...) è 0

```
Rettangolo rect = new Rettangolo();  
System.out.println(rect.getBase()); // Stampa 0
```

- Stampa (sempre e prevedibilmente) 0 nonostante non sia stato invocato il metodo **setBase(0);**

Variabili Locali: Inizializzazione (2)

- Le variabili locali non vengono però inizializzate automaticamente in Java
 - È necessario associargli un valore di inizializzazione al momento della loro dichiarazione, altrimenti si genera un errore già a tempo di compilazione
- Si evitano alla radice errori difficili da individuare (a tempo di esecuzione)
 - In C, le variabili non inizializzate assumono valori apparentemente casuali
 - Contenuto nelle zone di memoria *sporche* relative a tali variabili
 - Il programma potrebbe anche non generare errori ma il suo comportamento sarà comunque non prevedibile

Variabili Locali: Inizializzazione (3)


- In Java, per evitare questo problema è necessario associare *esplicitamente* un valore iniziale alle variabili locali
- Perché nel linguaggio C si è fatta la scelta opposta? Perché l'inizializzazione costa, in generale, e non sempre serve
 - Obbiettivi diversi dei due linguaggi (<<)
- Sebbene spesso non sia strettamente necessario esplicitamente assegnare un valore iniziale alle variabili di istanza, è buona consuetudine farlo

Variabili di Istanza: Campo d'Azione (o *Scope*)

- Le variabili di istanza sono visibili all'interno della sola classe in cui sono dichiarate
 - Ogni metodo può referenziarle semplicemente per nome

```
public class Rettangolo {  
    private int base;  
    // ...  
    public int getBase() {  
        return base;  
    }  
    // ...  
}
```

Qui 'base' fa
riferimento alla
variabile di
istanza **base**



Variabili di Istanza

- Una variabile di istanza ha un valore come parte dello stato di uno specifico oggetto istanza della sua classe
- Si usa dire che una variabile di istanza *“appartiene ad un oggetto”* anche se è *definita* nella sua classe
- Un oggetto, tramite le proprie variabili di istanza, possiede un proprio stato *autonomamente* rispetto a tutti gli altri oggetti istanza della sua stessa classe

~~Rettangolo.base; // NON COMPILA~~

```
Rettangolo rect1 = new Rettangolo();  
rect1.setBase(10);
```

```
Rettangolo rect2 = new Rettangolo();  
rect2.setBase(20); /* N.B. la base del secondo oggetto cambia;  
                    Quella del primo rimane invariata */
```

Un Parallelismo con il Linguaggio C (1)

- Pare abbastanza naturale associare una variabile di istanza di una classe Java ad un campo di una **struct** di C

```
typedef struct {  
    int base;  
    ...  
} Rettangolo;
```

```
Rettangolo *r = malloc(sizeof(Rettangolo));  
r->base = 15;  
Rettangolo *r2 = malloc(sizeof(Rettangolo));  
r2->base = 30;  
free(r1);  
free(r2);
```

Un Parallelismo con il Linguaggio C (2)

- I metodi definiscono le operazioni che si possono svolgere su un oggetto di una certa classe
- Viene naturale associare un metodo di una classe Java ad una funzione C che opera su una **struct**

```
typedef struct {  
    int base;  
    ...  
} Rettangolo;
```

```
void setBase(Rettangolo *this, int base) {  
    this->base = base;  
}
```

Invocazione dei Metodi

- L'invocazione dei metodi è alla base della programmazione orientata agli oggetti come meccanismo per lo scambio di messaggi tra oggetti
- I metodi mettono in comunicazione diretta l'oggetto che invoca il metodo con quello su cui il metodo viene invocato
- Ad esempio:
 - Per impostare od ottenere la base di un rettangolo abbiamo invocato dei metodi della classe **Rettangolo**
 - Per spostare un oggetto istanza della classe **Rettangolo** il suo metodo **sposta()** ha invocato dei metodi della classe **Punto** una cui istanza ne rappresenta il vertice

Metodo `main()`

- L'esecuzione di un programma inizia sempre con l'invocazione di un particolare e specifico metodo
- Per convenzione (eredità dal linguaggio C) tale metodo si chiama `main()`
 - Questo metodo “scatena” l'esecuzione invocando a sua volta altri metodi
- Tranne che per il metodo `main()` da cui comincia l'esecuzione, per ogni invocazione di metodo esiste sempre un metodo *invocante* ed un metodo *invocato*

Metodo Invocante e Invocato

Invocazione di metodo

*Metodo
invocante*

```
public class Main {  
    public static void main(String args[]) {  
        Rettangolo rect = new Rettangolo();  
        rect.setBase(22);  
    }  
}
```

*Metodo
invocato*

Definizione di Metodo

- I metodi sono dichiarati all'interno della definizione di una classe e definiscono il comportamento di tutti gli oggetti appartenenti a quella classe
- La dichiarazione di un metodo comprende due parti:
 - Intestazione
 - Modificatore di accesso/visibilità
 - Tipo valore restituito
 - Nome del metodo
 - Lista dei parametri formali
 - Corpo
 - Definizioni di variabili locali
 - Istruzioni

Intestazione *Corpo*

```
public void setX(int x) { ... }
```


Metodi: Valore Restituito

- I metodi possono comunicare verso l'esterno restituendo un valore
 - Il metodo *invocato* comunica con il metodo *invocante*
 - esattamente come per le funzioni in C
- Se un metodo non ritorna nessun valore al momento della dichiarazione del tipo di ritorno si utilizza la parola chiave **void**

Metodi e Aggiornamenti di Stato

- Conviene, per diversi motivi (>>), distinguere sempre i metodi che
 - **interrogano** (solamente) lo stato dell'oggetto su cui sono invocati
 - Solo *lettura* dello stato
 - **aggiornano** lo stato dell'oggetto su cui sono invocati
 - Anche *scrittura* dello stato
- Ad esempio (nella classe **Punto**)

```
public int getX() {  
    return x;           // interroga lo stato  
}  
  
public void trasla(int dx, int dy) {  
    x += dx;            // aggiorna lo stato  
    y += dy;            // aggiorna lo stato  
}
```

(Esercizio con Eclipse)

Variabili di Istanza e Metodi

- Realizzare la classe **Attrezzo**
 - Con le variabili di istanza
 - **nome** di tipo **String**
 - **peso** di tipo **int**
 - aggiungere i relativi metodi *getter & setter*
- Realizzare la classe **Stanza**
 - Con le variabili di istanza
 - **nome** di tipo **String**
 - **stanzaAdiacente** di tipo **Stanza**
 - **attrezzoContenuto** di tipo **Attrezzo**
 - aggiungere i relativi metodi *getter & setter*

Modificare lo Stato di un Oggetto (1)

- Lo stato di un oggetto può essere cambiato

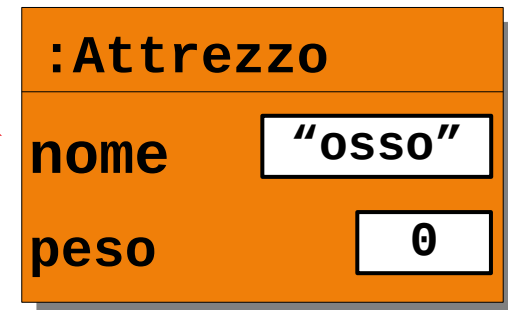
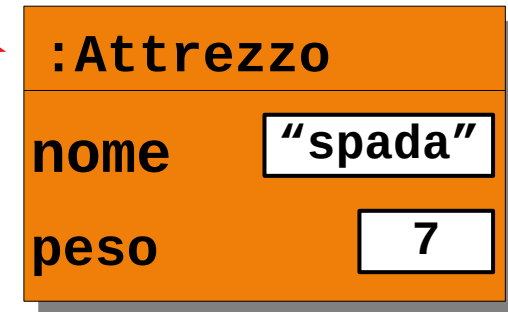
```
public class MainStanzeAttrezzi {  
    public static void main(String[] args) {  
        Attrezzo spada = new Attrezzo();  
        spada.setNome("spada");  
        spada.setPeso(7);  
  
        Attrezzo osso = new Attrezzo();  
        Osso.setNome("osso");  
        Osso.setPeso(1);  
  
        Stanza n11 = new Stanza();  
        n11.setNome("N11");  
  
        n11.setAttrezzo(spada);  
  
        n11.setAttrezzo(osso);  
    }  
}
```



Modificare lo Stato di un Oggetto (2)

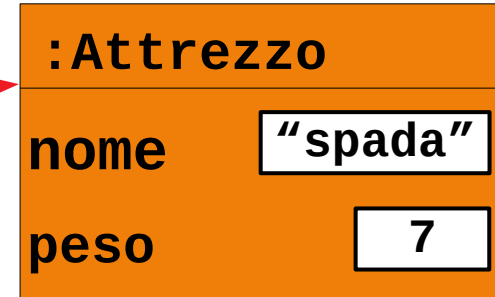
- Lo stato di un oggetto può essere cambiato

```
public class MainStanzeAttrezzi {  
    public static void main(String[] args) {  
        Attrezzo spada = new Attrezzo();  
        spada.setNome("spada");  
        spada.setPeso(7);  
  
        Attrezzo osso = new Attrezzo();  
        osso.setNome("osso");  
        osso.setPeso(1);  
  
        Stanza n11 = new Stanza();  
        n11.setNome("N11");  
  
        n11.setAttrezzo(spada);  
  
        n11.setAttrezzo(osso);  
    }  
}
```

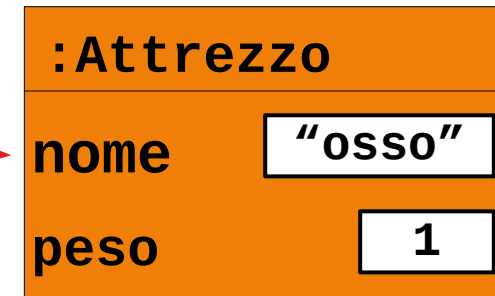


Modificare lo Stato di un Oggetto (3)

```
public class MainStanzeAttrezzi {  
    public static void main(String[] args) {  
        Attrezzo spada = new Attrezzo();  
        spada.setNome("spada");  
        spada.setPeso(7);
```



```
        Attrezzo osso = new Attrezzo();  
        osso.setNome("osso");  
        osso.setPeso(1);
```



```
        Stanza n11 = new Stanza();  
        n11.setNome("N11");
```

```
        n11.setAttrezzo(spada);
```

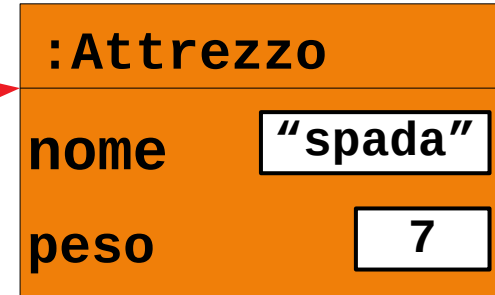
```
        n11.setAttrezzo(osso);
```

```
    }
```

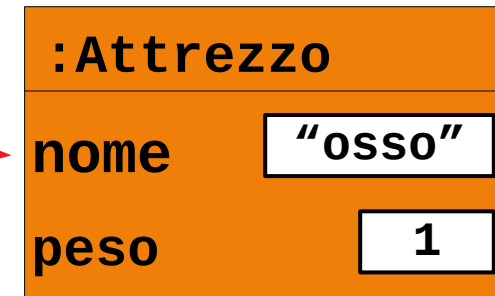
```
}
```

Modificare lo Stato di un Oggetto (4)

```
public class MainStanzeAttrezzi {  
    public static void main(String[] args) {  
        Attrezzo spada = new Attrezzo();  
        spada.setNome("spada");  
        spada.setPeso(7);
```



```
        Attrezzo osso = new Attrezzo();  
        Osso.setNome("osso");  
        Osso.setPeso(1);
```



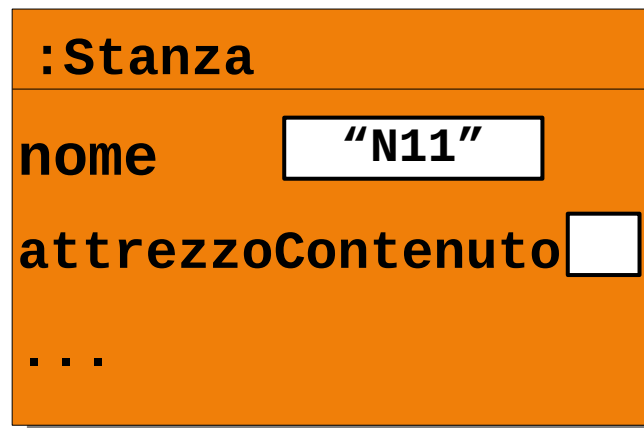
```
        Stanza n11 = new Stanza();  
        n11.setNome("N11");
```

```
        n11.setAttrezzo(spada);
```

```
        n11.setAttrezzo(osso);
```

```
    }
```

```
}
```



Modificare lo Stato di un Oggetto (5)

```
public class MainStanzeAttrezzi {  
    public static void main(String[] args) {  
        Attrezzo spada = new Attrezzo();  
        spada.setNome("spada");  
        spada.setPeso(7);
```

```
        Attrezzo osso = new Attrezzo();  
        Osso.setNome("osso");  
        Osso.setPeso(1);
```

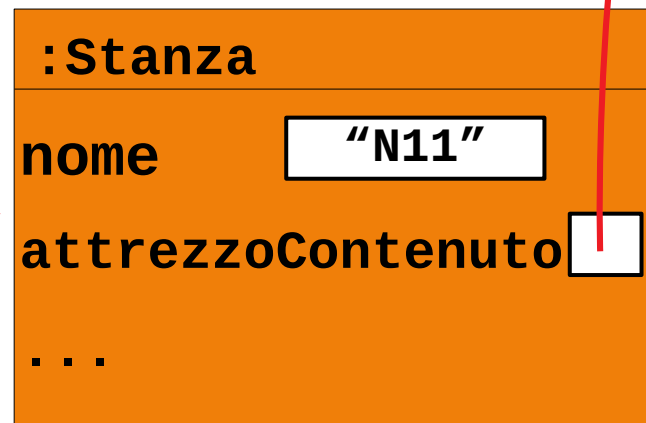
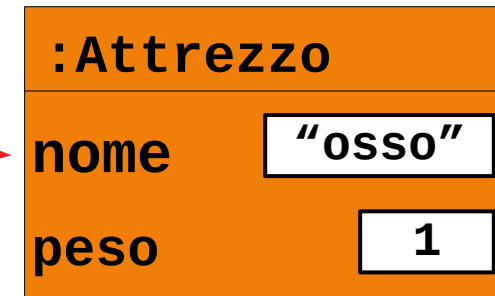
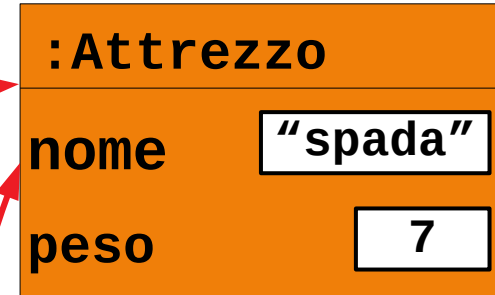
```
        Stanza n11 = new Stanza();  
        n11.setNome("N11");
```

```
        n11.setAttrezzo(spada);
```

```
        n11.setAttrezzo(osso);
```

```
    }
```

```
}
```



Modificare lo Stato di un Oggetto (6)

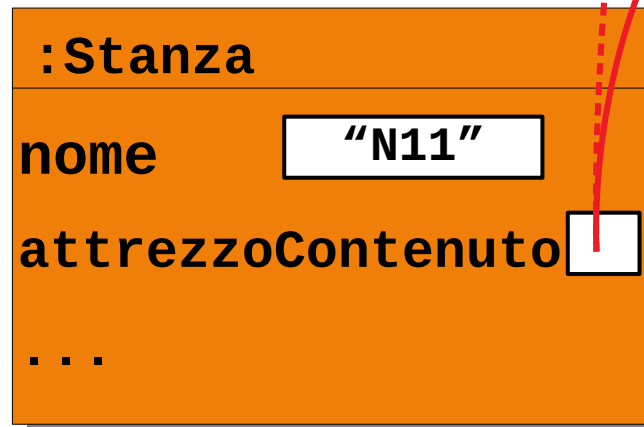
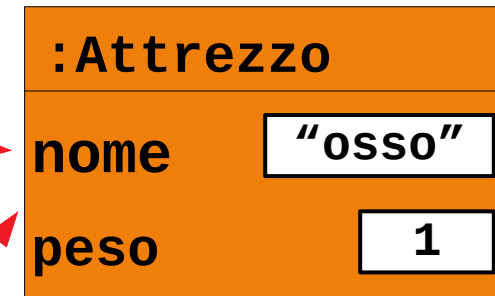
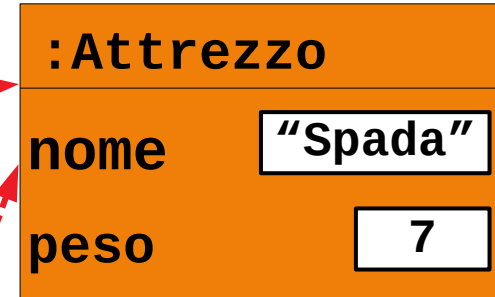
```
public class MainStanzeAttrezzi {  
    public static void main(String[] args) {  
        Attrezzo spada = new Attrezzo();  
        spada.setNome("spada");  
        spada.setPeso(7);
```

```
        Attrezzo osso = new Attrezzo();  
        Osso.setNome("osso");  
        Osso.setPeso(1);
```

```
        Stanza n11 = new Stanza();  
        n11.setNome("N11");
```

```
        n11.setAttrezzo(spada);
```

```
        n11.setAttrezzo(osso);  
    }  
}
```



Metodi: Parametri (1)

- I metodi possono ricevere dati mediante il passaggio di parametri in ingresso
- La lista dei parametri è dichiarata nell'intestazione del metodo. Ad es.:

```
public void setXY(int nuovaX, int nuovaY) {  
    x = nuovaX;  
    y = nuovaY;  
}
```

- Metodo di due parametri
 - nuovaX e nuovaY
entrambi di tipo `int`

Metodi: Parametri (2)

- Distinguiamo
 - Parametri *formali*
 - Ad indicare i parametri nella definizione di un metodo
 - ✓ Concetto a *tempo statico* (durante la compilazione)
 - Parametri *attuali* (o *argomenti*)
 - Per indicare i parametri che effettivamente vengono passati ad un metodo all'atto della sua invocazione
 - ✓ Concetto a *tempo dinamico* (durante l'esecuzione)
- ✓ Non è niente affatto casuale che il concetto di parametro formale *sta* a quello di argomento come il concetto di classe *sta* a quello di oggetto

Passaggio dei Parametri

- A differenza che nel linguaggio di programmazione C, il passaggio dei parametri nel linguaggio Java può avvenire *solo per valore*
- ✓ Il corpo di un metodo lavora su una *copia* distinta ed autonoma ma identica nel contenuto all'argomento ricevuto al momento dell'invocazione

```
public class ModificatoreValori {  
    public void azzera(int v) {  
        v = 0; // v è una copia, sono variabili distinte  
    }  
  
    public static void main(String[] args) {  
        int valore = 5;  
        ModificatoreValori mod = new ModificatoreValori();  
        System.out.println(valore); // Stampa 5;  
        mod.azzera(valore);  
        System.out.println(valore); // Stampa 5;  
    }  
}
```

Variabili Locali ai Metodi

- All'interno dei metodi è possibile dichiarare delle variabili locali
- Servono a memorizzare informazioni utili e di supporto all'esecuzione del metodo
- *Ciclo di vita* limitato al più alla durata dell'esecuzione del corpo del metodo
 - create dopo la dichiarazione della variabile
 - distrutte quando il metodo termina
- La *visibilità* (scope) di una variabile locale è limitata al blocco di codice (ovvero entro {...}) nel quale è definita
- Spesso (ma non necessariamente) l'intero corpo del metodo
 - Non può essere acceduta da altri metodi
 - Né della stessa classe
 - Né di altre classi

Variabili Locali: Esempio

- Un esempio già visto fa uso di variabili locali:
 - Il metodo `sposta()` in `Rettangolo` quando la classe `Punto` non dispone del metodo `trasla()`

```
public class Rettangolo {  
    // ... come prima ...  
    public void sposta(int deltaX, int deltaY) {  
        int xVertice = vertice.getX();  
        int yVertice = vertice.getY();  
        vertice.setX(xVertice + deltaX);  
        vertice.setY(yVertice + deltaY);  
    }  
}
```

Variabili Locali, Precisazioni

- Una variabile locale può essere acceduta nel blocco di codice (racchiuso entro {...}) in cui viene dichiarata *solo successivamente* alla sua dichiarazione
- Può essere acceduta anche in tutti i blocchi di codice strettamente contenuti al suo interno
- Un blocco di codice più *esterno* certamente non può accedere ad una variabile dichiarata in un blocco di codice più *interno*
- Ad esempio, il seguente codice non compila:

```
{ int esterna = 0,  
  interna++;  
  { int interna = 0; }  
}
```

- In C? Dipende dal compilatore, dalla versione, e dallo standard adottato...

Variabili Locali, Precisazioni

- In Java è possibile dichiarare le variabili locali ovunque all'interno di un blocco di codice

```
{  
    int addendo1 = 1;  
    <istruzioni>  
    int addendo2 = 2;  
    <istruzioni>  
    int somma = addendo1 + addendo2;  
}
```

- In C? *Dipende!* Oggi il comportamento è sempre più diffusamente simile a quello di Java.

In passato no

Variabili Locali vs Variabili di Istanza

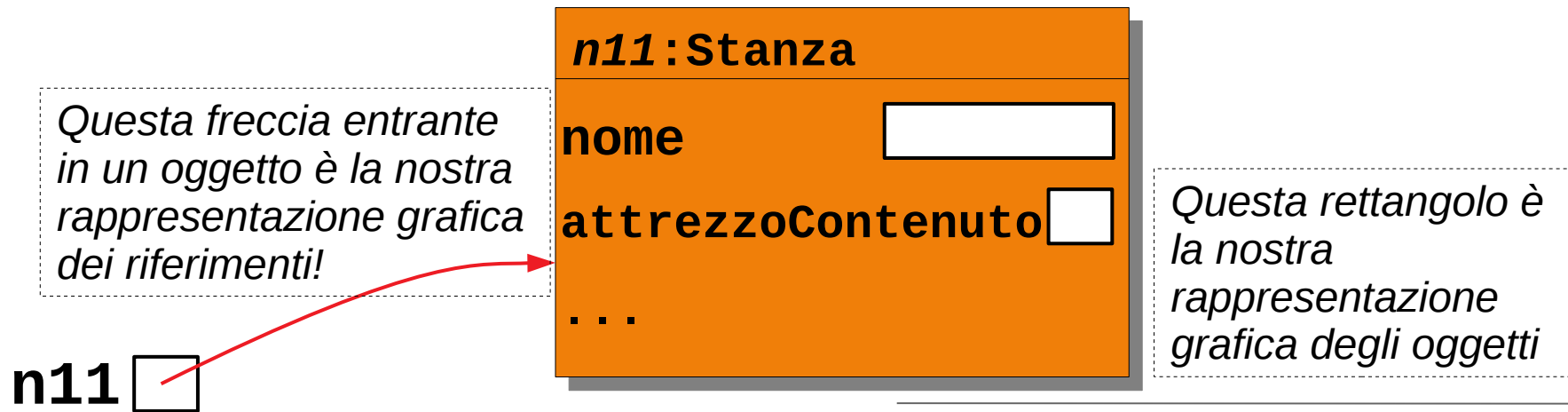
- Attenzione!

Le differenze tra una variabile locale e una variabile di istanza sono enormi ed importanti

- A cominciare dal ciclo di vita completamente diverso:
 - Le *variabili locali* devono memorizzare informazioni che servono esclusivamente alla esecuzione di un metodo
 - create durante l'esecuzione del metodo
 - distrutte al termine di questo
 - non sono visibili al di fuori del corpo del metodo
 - Le *variabili di istanza* memorizzano informazioni che rappresentano lo stato dell'oggetto
 - vivono per tutta la vita dell'oggetto
 - sono visibili a tutti i metodi della classe
 - è possibile, e consigliabile (>>), renderle non visibili al di fuori del corpo della classe
- È un errore molto grave confondere questi due concetti!

Riferimenti ad Oggetti (1)

- La creazione di un nuovo oggetto in memoria avviene tramite l'operatore **new**
- L'operatore **new** restituisce un *riferimento ad un oggetto* appena creato
- Ad esempio: **Stanza n11 = new Stanza();**
- La variabile locale **n11** **NON** contiene l'oggetto creato, ma bensì un *riferimento* ad esso (>>)



Riferimenti ad Oggetti (2)

- Proviamo a stampare il valore di variabili che contengono riferimenti

```
public class MainRiferimenti {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
        System.out.println(n11);  
    }  
}
```

- Stampa: **Stanza@15db9742**
 - Ma ovviamente dipende dalla particolare esecuzione
 - Possiamo per il momento semplificare il significato di questa stampa: è *[un numero che dipende dal]l'indirizzo in memoria dell'oggetto referenziato*
 - In realtà non è esattamente così, ma per i nostri presenti scopi questa semplificazione fa molto comodo (>>)

Riferimenti ad Oggetti (3)

- Una *stessa* variabile può contenere, in momenti diversi, riferimenti ad oggetti distinti dello stesso tipo. Ad esempio:

```
public class MainRiferimenti {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
        System.out.println(n11); // Stampa Stanza@15db9742  
  
        n11 = new Stanza();  
        System.out.println(n11); // Stampa Stanza@6d06d69c  
    }  
}
```

Riferimenti ad Oggetti (4)

```
public class MainRiferimenti {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
        System.out.println(n11); // stampa Stanza@15db9742  
  
        n11 = new Stanza();  
        System.out.println(n11); // stampa Stanza@6d06d69c  
    }  
}
```

n11 ☐

0x15db9742:Stanza

nome

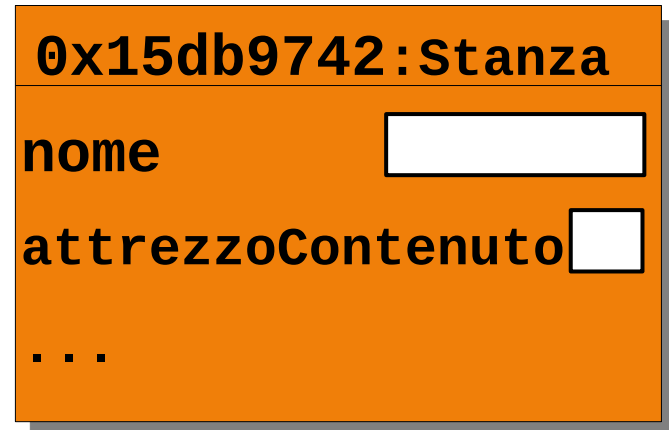
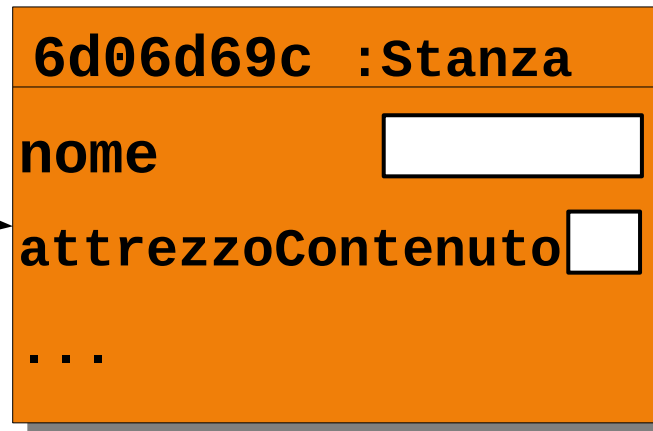
attrezzoContenuto

...

Riferimenti ad Oggetti (5)

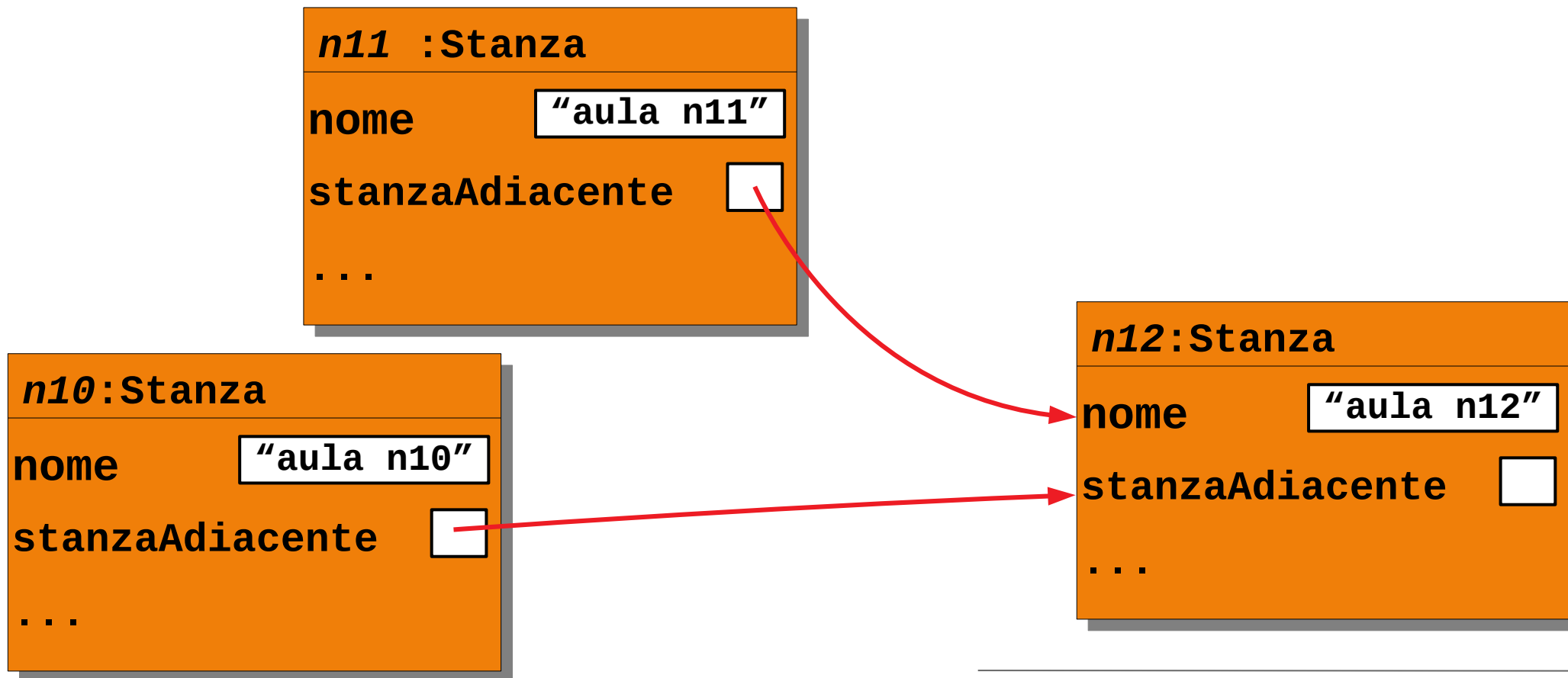
```
public class MainRiferimenti {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
        System.out.println(n11); // stampa Stanza@15db9742  
  
        n11 = new Stanza();  
        System.out.println(n11); // stampa Stanza@6d06d69c  
    }  
}
```

n11 ☐



Molteplici Riferimenti verso lo Stesso Oggetto (1)

- In alcuni casi più variabili contengono un riferimento allo stesso oggetto
- Ad esempio due stanze adiacenti la medesima:



Molteplici Riferimenti verso lo Stesso Oggetto (2)

- La configurazione appena vista si può ottenere con il seguente codice

```
public class MainStanzeRiferimenti {  
    public static void main(String[] args) {  
        Stanza n12 = new Stanza();  
        n12.setNome("aula n12");  
  
        Stanza n11 = new Stanza();  
        n11.setNome("aula n11");  
        n11.setStanzaAdiacente(n12);  
  
        Stanza n10 = new Stanza();  
        n10.setNome("aula n10");  
        n10.setStanzaAdiacente(n12);  
    }  
}
```


Molteplici Riferimenti verso lo Stesso Oggetto (3)

- Un altro esempio:

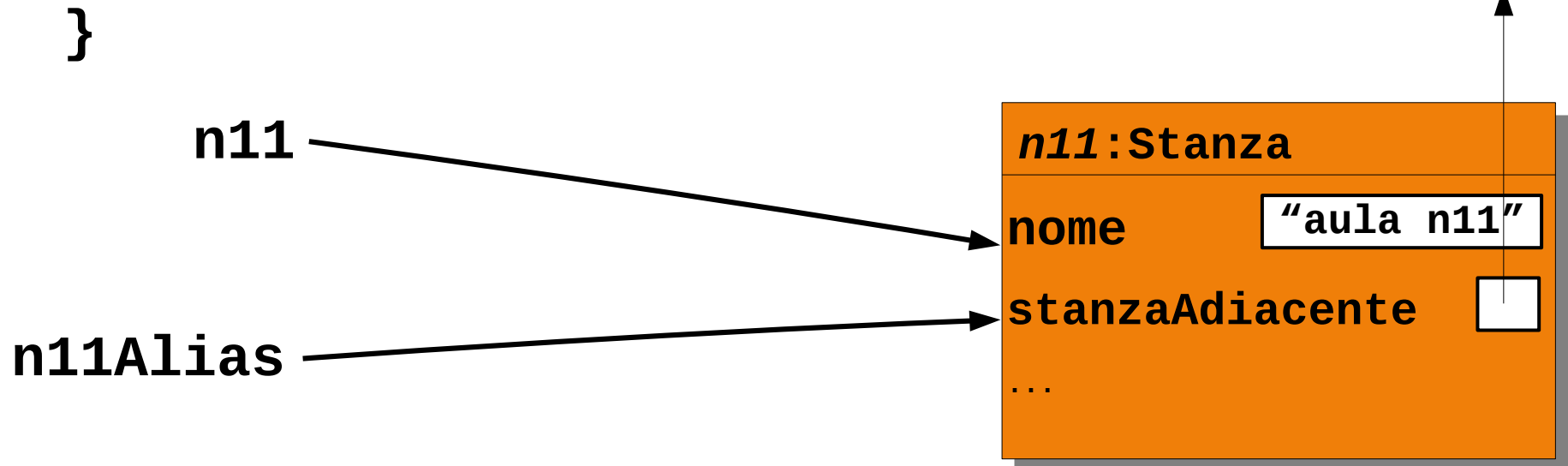
```
public class MainStanzeRiferimenti {  
    public static void main(String[] args) {  
        Stanza n12 = new Stanza();  
        n12.setNome("aula n12");  
  
        Stanza n11 = new Stanza();  
        n11.setNome("aula n11");  
        n11.setStanzaAdiacente(n12);  
  
        Stanza n11Alias = n11;  
    }  
}
```

- Ora sia **n11** sia **n11Alias** fanno riferimento allo stesso oggetto

Molteplici Riferimenti verso lo Stesso Oggetto (4)

```
public class MainStanzeRiferimenti {  
    public static void main(String[] args) {  
        Stanza n12 = new Stanza();  
        n12.setNome("aula n12");  
  
        Stanza n11 = new Stanza();  
        n11.setNome("aula n11");  
        n11.setStanzaAdiacente(n12);  
  
        Stanza n11Alias = n11;  
    }  
}
```

Stanza n11Alias = n11;



Più Riferimenti & Side-Effect (1)

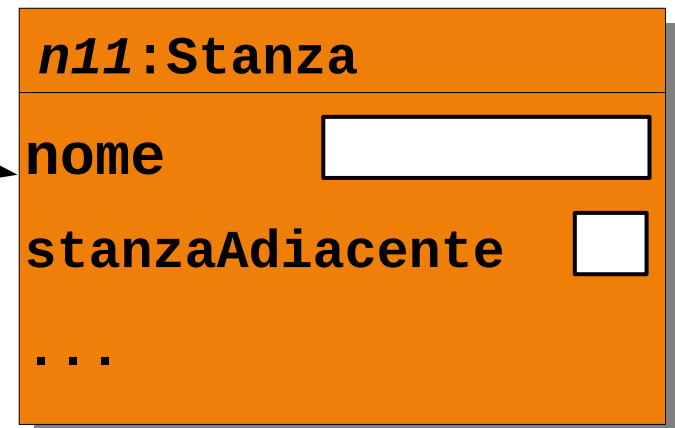
- Qual è l'output del seguente programma?

```
public static MainRiferimentiSideEffect {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
  
        Stanza n11Alias = n11;  
  
        n11.setNome("N11");  
        n11Alias.setNome("aula N11");  
  
        System.out.println(n11.getNome());  
    }  
}
```

Più Riferimenti & Side-Effect (2)

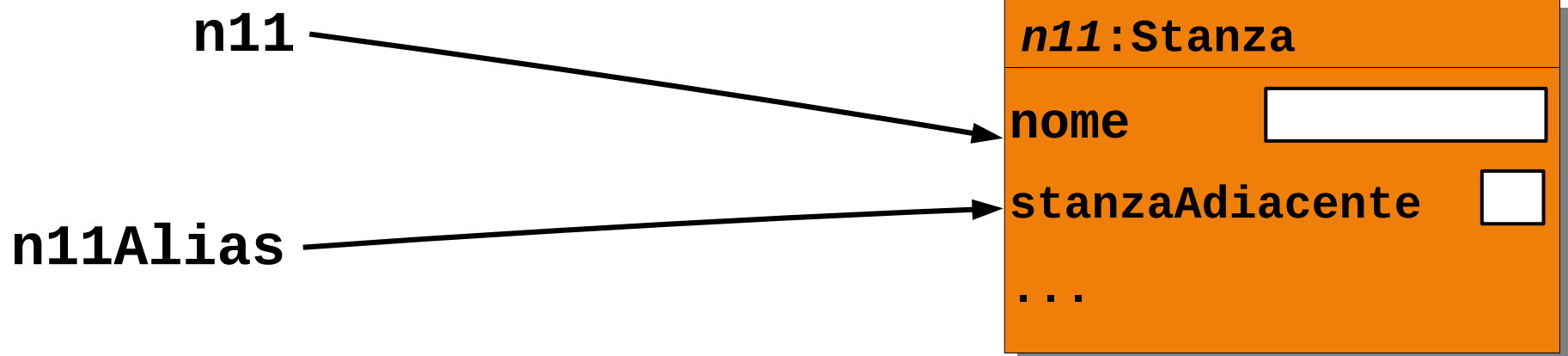
```
public static MainRiferimentiSideEffect {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
        Stanza n11Alias = n11;  
  
        n11.setNome("N11");  
        n11Alias.setNome("aula N11");  
  
        System.out.println(n11.getNome());  
    }  
}
```

n11



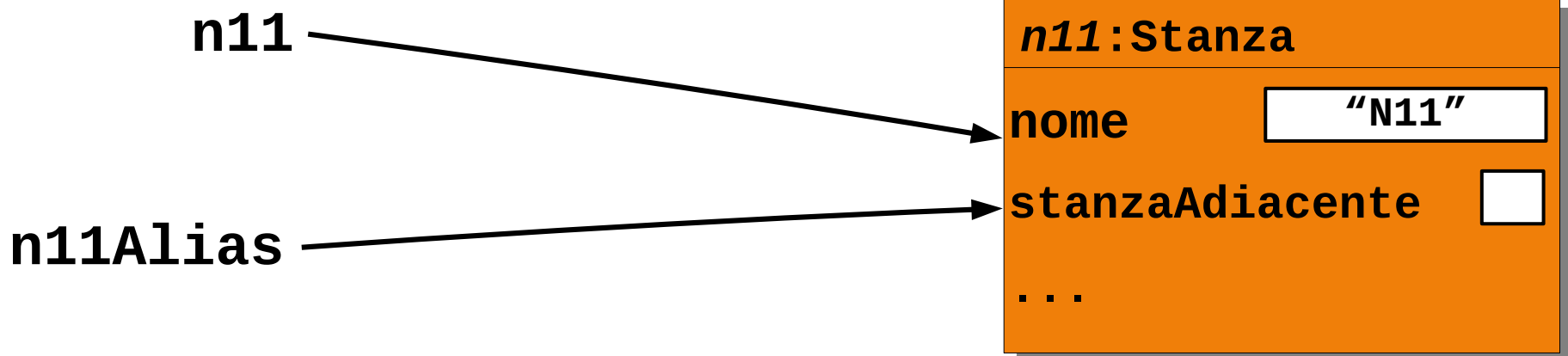
Più Riferimenti & Side-Effect (3)

```
public static MainRiferimentiSideEffect {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
  
        Stanza n11Alias = n11;  
  
        n11.setNome("N11");  
        n11Alias.setNome("aula N11");  
  
        System.out.println(n11.getNome());  
    }  
}
```



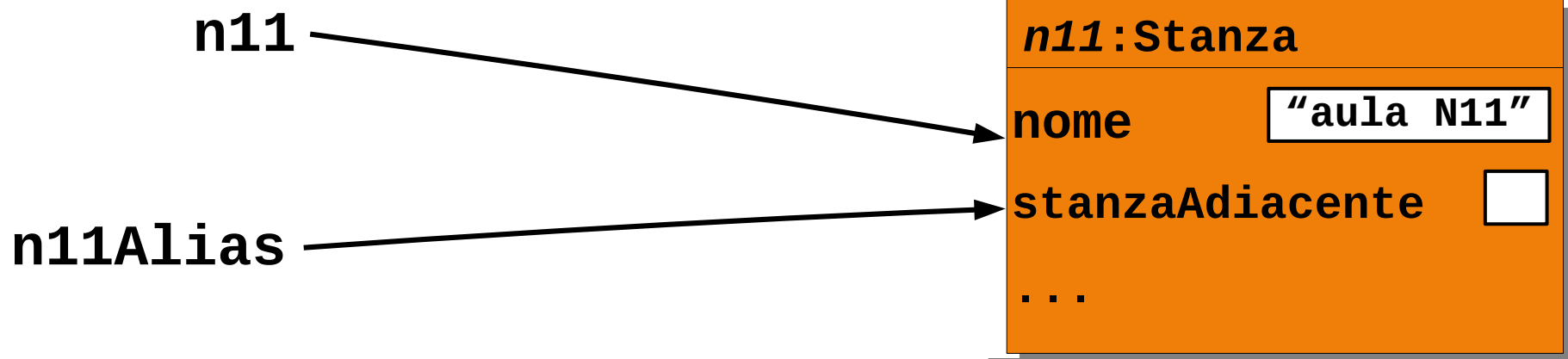
Più Riferimenti & Side-Effect (4)

```
public static MainRiferimentiSideEffect {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
  
        Stanza n11Alias = n11;  
  
        n11.setNome("N11");  
        n11Alias.setNome("aula N11");  
  
        System.out.println(n11.getNome());  
    }  
}
```



Più Riferimenti & Side-Effect (5)

```
public static MainRiferimentiSideEffect {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
  
        Stanza n11Alias = n11;  
  
        n11.setNome("N11");  
        n11Alias.setNome("aula N11");  
        System.out.println(n11.getNome());  
    }  
}
```



Più Riferimenti & Side-Effect (6)

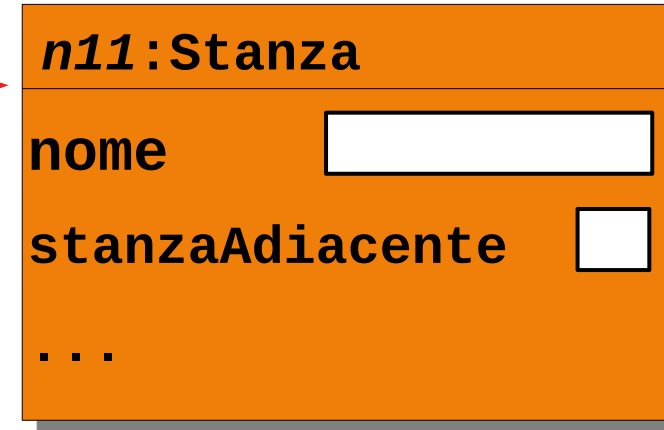
- L'output è “**aula N11**”
- Sorprendente per chi aveva creato l'oggetto e lo aveva chiamato semplicemente “**n11**”?
- Questo tipo di comportamenti spesso vengono indicati con il nome di *Effetti Collaterali (Side-Effect)*
 - un'azione genera effetti visibili ben al di fuori dell'ambito in cui è avvenuta
- Sia la variabile **n11** che **n11Alias** fanno riferimento allo stesso oggetto
 - una modifica effettuata a tale oggetto tramite uno dei due riferimenti è visibile *anche* usando l'altro

Riferimenti & Parametri per Valore

- Quando una variabile contenente un riferimento è passata come argomento ad un metodo
 - il passaggio è per valore
 - viene copiato *il riferimento* contenuto nell'argomento
 - l'oggetto a cui fa riferimento *NON* viene copiato
 - Similarmente a quanto avviene, in C, passando (per valore) il puntatore ad aree di **memoria** allocate con **malloc**
 - è possibile cambiare il contenuto della **memoria** il cui **indirizzo** è fornito come argomento
 - non è possibile cambiare il contenuto della variabile che conteneva tale **indirizzo** al momento dell'invocazione
- anche in Java, passando un **riferimento** ad un **oggetto**
- è possibile cambiare lo stato dell'**oggetto** il cui **riferimento** è fornito come argomento
 - non è possibile cambiare il contenuto della variabile che conteneva tale **riferimento** al momento dell'invocazione

Passaggio di Riferimenti (1)

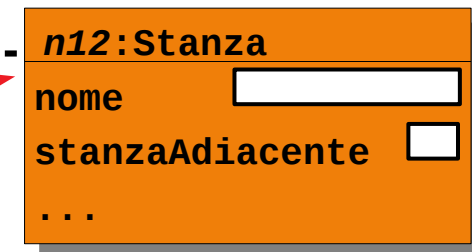
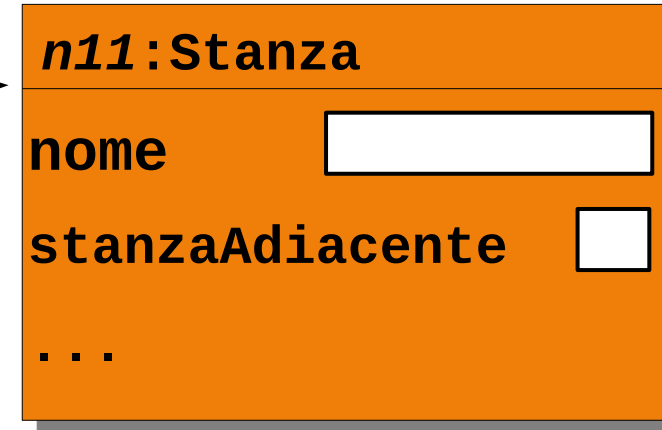
```
public class MainPassRef {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
        Stanza n12 = new Stanza();  
        n11.setStanzaAdiacente(n12);  
    }  
}
```



```
public class Stanza {  
    // ...  
    public void setStanzaAdiacente(Stanza stanza) {  
        this.stanzaAdiacente = stanza;  
    }  
}
```

Passaggio di Riferimenti (2)

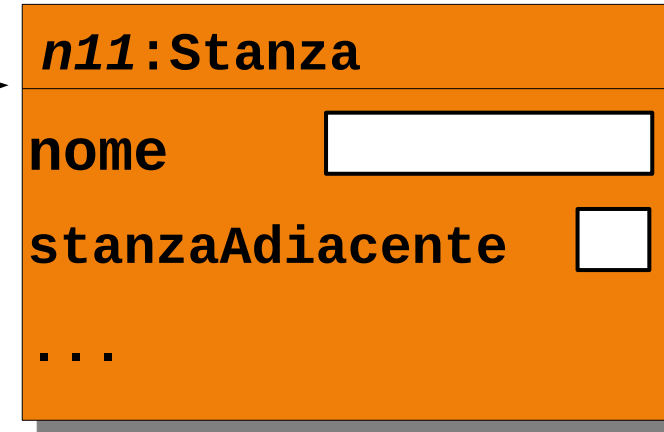
```
public class MainPassRef {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
  
        Stanza n12 = new Stanza();  
        n11.setStanzaAdiacente(n12);  
    }  
}
```



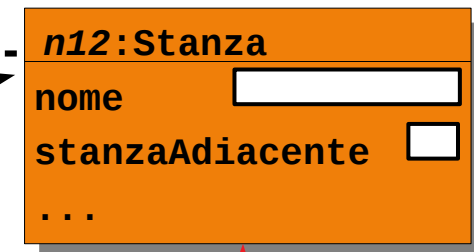
```
-----  
public class Stanza {  
    // ...  
  
    public void setStanzaAdiacente(Stanza stanza) {  
        this.stanzaAdiacente = stanza;  
    }  
}
```

Passaggio di Riferimenti (3)

```
public class MainPassRef {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
  
        Stanza n12 = new Stanza();  
        n11.setStanzaAdiacente(n12);  
    }  
}
```

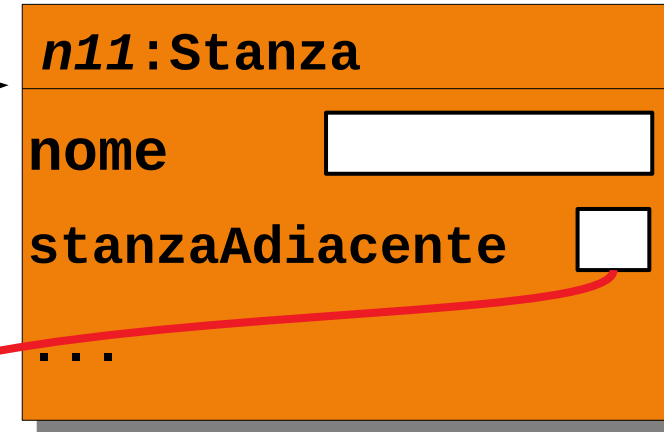


```
public class Stanza {  
    // ...  
    public void setStanzaAdiacente(Stanza stanza) {  
        this.stanzaAdiacente = stanza;  
    }  
}
```

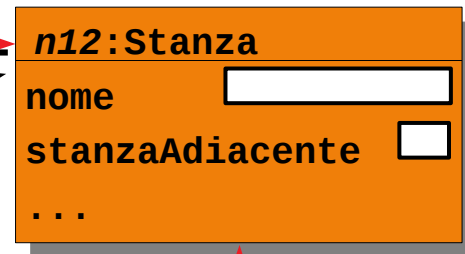


Passaggio di Riferimenti (4)

```
public class MainPassRef {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
  
        Stanza n12 = new Stanza();  
        n11.setStanzaAdiacente(n12);  
    }  
}
```



```
public class Stanza {  
    // ...  
    public void setStanzaAdiacente(Stanza stanza) {  
        this.stanzaAdiacente = stanza;  
    }  
}
```



Riferimenti & Valore Restituito

- Quando un riferimento viene *restituito* da un metodo
 - Viene restituita una *copia del riferimento*
 - L'oggetto a cui si riferisce *NON* viene copiato

```
public class Stanza {  
    // ...  
    public Stanza getStanzaAdiacente() {  
        return stanzaAdiacente;  
    }  
}
```

Esercizio (*con Eclipse*)

- Assumiamo che
 - la classe **Rettangolo** non disponga del metodo **sposta()**
 - la classe **Punto** invece disponga del metodo **trasla()**
- Trovare un modo alternativo per spostare gli oggetti **Rettangolo**
- Vediamo due soluzioni:
N.B. nessuna delle due è raccomandabile (>>)

Esercizio (2)

- Prima soluzione

```
public static void main(String[] args) {  
    Punto origine = new Punto();  
    origine.setX(0);  
    origine.setY(0);  
    Rettangolo rect = new Rettangolo();  
    rect.setVertice(origine);  
  
    origine.trasla(1, 1);  
}
```

- Se spostiamo l'oggetto istanza della classe **Punto** che utilizziamo come vertice dell'oggetto istanza della classe **Rettangolo**, spostiamo, *come effetto collaterale*, il rettangolo stesso

Esercizio (3)

- Seconda soluzione

```
public static void main(String[] args) {  
    Punto origine = new Punto();  
    origine.setX(0);  
    origine.setY(0);  
    Rettangolo rect = new Rettangolo();  
    rect.setVertice(origine);  
  
    Punto verticeRect = rect.getVertice();  
    verticeRect.trasla(1, 1);  
}
```

- Si ottiene *una copia del riferimento* all'oggetto istanza della class **Punto** che figura come vertice dell'oggetto istanza della classe **Rettangolo** che spostato produce, *ancora come effetto collaterale*, lo spostamento del rettangolo stesso

Esercizio (4)

- Entrambe le soluzioni sono poco raccomandabili, perché non rendono affatto evidente la reale intenzione di spostare il rettangolo
- Nessuna *invocazione diretta* di un metodo della classe **Rettangolo** induce a pensare che lo si sta spostando
- Il rettangolo viene spostato solamente come il risultato dell'effetto collaterale dello spostamento di un punto di cui conservava un riferimento
- E' preferibile dotare la classe **Rettangolo** di un apposito metodo **trasla()**
- Gli effetti collaterali risultano difficili da tracciare

Riferimento *Null*

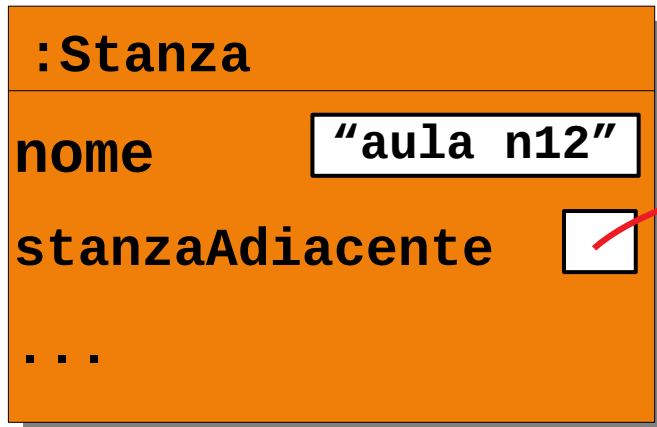
- In Java esiste un solo letterale di tipo riferimento ad oggetto: **null**
- Un valore speciale e distinto da tutti gli altri valori, il riferimento *null*
- Indica l'assenza di un reale riferimento ad un oggetto esistente
 - N.B. In Java non esiste alcuna relazione particolare tra il valore **null** e 0 (letterale di tipo **int**)
 - In C, la macro **NULL** è invece un *alias* per il valore 0
- Un po' come già accadeva per i booleani, la tipizzazione Java è più *stringente*

Utilizzo del Riferimento Nullo

- Il riferimento nullo è utile in vari contesti
 - come valore speciale restituito da un metodo per segnalare un caso speciale. Ad es.:
Persona cercata = rubrica.trova("alice");
restituisce **null** se non esiste alcuna persona di nome "alice" nella rubrica
 - per fornire un valore di default a variabili che contengono riferimenti ad oggetti

Uso di null: Esempio

```
public class MainNull {  
    public static void main(String[] args) {  
        Stanza n12 = new Stanza();  
        n12.setNome("aula n12");  
  
        n12.setStanzaAdiacente( null );  
    }  
}
```



Si intende
rappresentare che
la stanza **n12** *NON*
possiede stanze
adiacenti

NullPointerException (1)

- Cosa succede se si invoca un metodo su un riferimento nullo?

```
public class MainNullPointerException {  
    public static void main(String[] args) {  
        Stanza n12 = new Stanza();  
        n12.setNome("aula n12");  
  
        n12.setStanzaAdiacente( null );  
  
        Stanza adiacenteN12 = n12.getStanzaAdiacente();  
  
        System.out.println(adiacenteN12.getNome());  
    }  
}
```

NullPointerException (2)

- **null** rappresenta l'assenza di un riferimento ad un oggetto
 - Si genera un errore a tempo di esecuzione, un'eccezione (*runtime-exception* >>)

NullPointerException

```
$ java MainNullPointerException
```

```
Exception in thread "main"
```

```
java.lang.NullPointerException
```

```
at MainNullPointerException.main(MainNullPointerException.java:10)
```

*Tipologia
eccezione*

Numero di linea del codice in cui si è verificata

NullPointerException: Diagnostica

- Java, ancora una volta, fornisce una diagnostica efficace
- Cosa accadrebbe utilizzando il linguaggio di programmazione C?

```
struct Punto {  
    int x;  
    int y;  
};  
int main() {  
    struct Punto *origine = NULL;  
    origine->x = 0;  
}
```

Segmentation fault!

Inizializzazione delle Variabili di Istanza e null

- Il compilatore forza l'inizializzazione di tutte le variabili di istanza
- Quelle dichiarate come contenenti un riferimento ad oggetto sono inizializzate a **null**

```
Rettangolo rect = new Rettangolo();  
System.out.println(rect.getBase()); // 0  
System.out.println(rect.getVertice()); // null  
System.out.println(rect.getVertice().getX());
```

✓ Il valore restituito da **getVertice()** è **null**,
invocando un metodo sul suo risultato si genera
una **NullPointerException**

Evitare NullPointerException

- Se è noto che una funzione può ritornare **null** come valore speciale è necessario predisporre un controllo sul valore restituito

...

```
Persona cercata = rubrica.trova("alice");  
if (cercata!=null)  
    System.out.println(cercata.getEta());  
else  
    System.out.println("non trovato");
```

- In C: **if (cercata!=0)**
 - In Java non compilerebbe

Campo d'Azione delle Variabili e dei Parametri

- Se il metodo `setX()` venisse così dichiarato?

```
public class Punto {  
    private int x;  
    private int y;  
    public void setX(int x) {  
        x = x;  
    }  
    ...  
}
```

```
Punto unoUno = new Punto();  
unoUno.setX(1);  
System.out.println(unoUno.getX()); // Stampa 0
```

Shadowing

- Si è verificato il cosiddetto *shadowing*:
 - Il parametro formale `x` ha lo stesso nome della variabile di istanza `x`
 - Il parametro formale ha però uno scope ($>>$) più ristretto e quindi ha precedenza
 - Nel contesto del corpo del metodo, l'identificatore '`x`' viene considerato un riferimento al parametro formale (e *non* alla var. di istanza)
 - Si dice anche che il parametro formale offusca ("fa ombra") la variabile di istanza
 - `x = x;` è un'espressione che assegna al parametro formale `x` il suo stesso valore (inutile!)
- ✓ Alcuni IDE moderni (come Eclipse) possono essere configurati per segnalare il problema

La Parola Chiave **this** (1)

- All'interno di ogni metodo è possibile ottenere un riferimento all'oggetto *corrente* utilizzando la parola chiave **this**
 - riferimento all'oggetto sul quale il metodo è stato invocato
 - tramite questo riferimento è quindi possibile:
 - modificare le variabili di istanza dell'oggetto
 - fare invocazioni di metodo nidificate sullo stesso oggetto
 - passare un riferimento all'oggetto corrente come argomento di altre invocazioni di metodo...

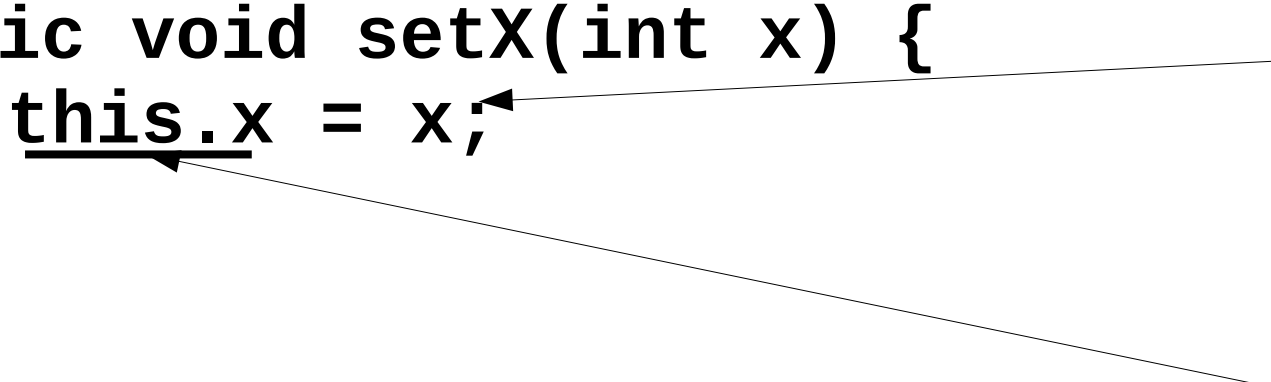
La Parola Chiave `this` (2)

- Si risolve anche il problema dello *shadowing*

```
...  
public void setX(int x) {  
    this.x = x;  
}  
...
```

Parametro

Variabile di istanza



```
Punto unoUno = new Punto();  
unoUno.setX(1);
```

```
System.out.println(unoUno.getX()); // Stampa 1
```

- All'interno del corpo del metodo `setX()`, `this` è un riferimento allo stesso oggetto a cui si riferisce anche `unoUno`

this in C?

- Talvolta può far comodo pensare a **this** come ad un parametro aggiuntivo passato automaticamente (ed implicitamente) ad ogni metodo
- Ad esempio il *metodo* **setX()** verrebbe tradotto in C con la seguente *funzione*

```
void setX(struct Punto *this, int x) {  
    this -> x = x;           // Codice C  
}
```

- Quindi l'invocazione di **unoUno.setX(1);** diverrebbe:

```
struct Punto unoUno;    // Codice C  
setX(&unoUno, 1);      // Codice C
```

Accedere le Variabili di Istanza con **this**

- **this** può essere usato per accedere alle variabili di istanza ma può anche essere omesso in assenza di ambiguità

```
...  
    public int getX() {  
        return this.x;  
    }  
...
```

- Equivale a

```
...  
    public int getX() {  
        return x;  
    }  
...
```

- Il compilatore risolve l'identificatore **x** come variabile di istanza dell'oggetto su cui il metodo è stato invocato
- In un certo senso, è come se aggiungesse **this.** automaticamente

this: Convenzione di Stile

- Adottiamo comunque la convenzione di usare *sempre e comunque* **this** per referenziare variabili di istanza
- Con le seguenti motivazioni:
 - si evita lo *shadowing*
 - si favorisce la leggibilità del codice
 - ✓ si favorisce l'apprendimento di questi concetti di base

Invocare Metodi Mediante **this** (1)

- La parola chiave **this** può essere usata per invocare metodi sullo stesso oggetto su cui il metodo corrente è stato invocato
 - Se **this** viene omissso il compilatore lo considera comunque presente
- Ad esempio è possibile scrivere il metodo **setXY()** usando i metodi **setX()** e **setY()**

Invocare Metodi Mediante `this` (2)

```
public void setXY(int x, int y) {  
    this.setX(x);  
    this.setY(y);  
}
```

- Anche per *alcune* invocazioni di metodi è consigliato usare **this** per aumentare la leggibilità
- Non sempre, ma quando si vuole evidenziare l'utilizzo di altri metodi della stessa classe nella scrittura di un primo metodo (passo *top-down*)

Esercizio

Fare le verifiche disponibili sul sito del corso:

- **Studente.java**
- **Tesi.java**
- **Sommatore.java**