

Programmazione Orientata agli Oggetti

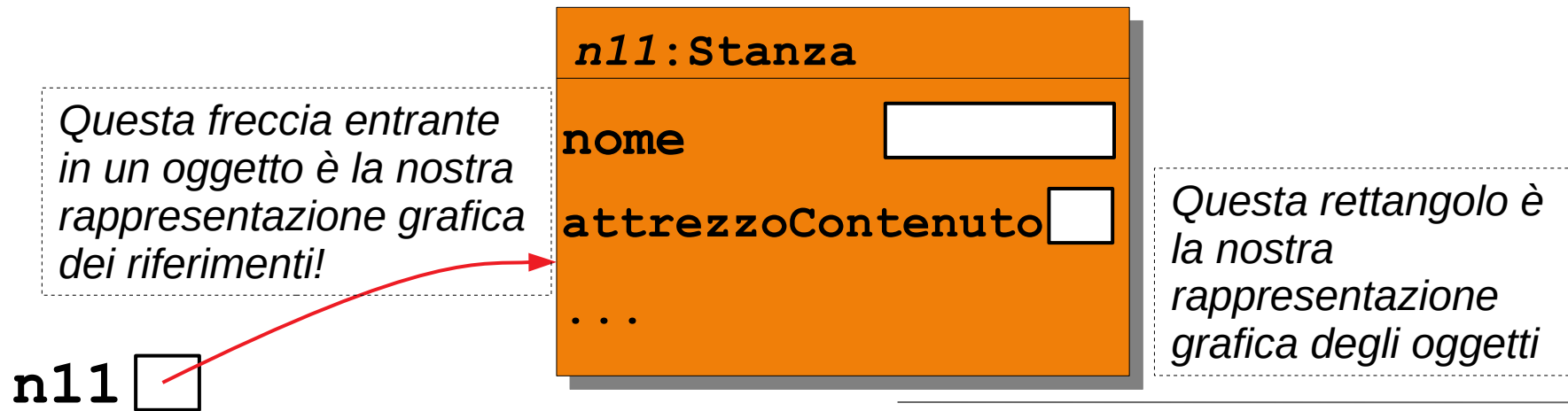
Oggetti e Riferimenti

Sommario

- Riferimenti ad Oggetti
 - Molteplici riferimenti verso lo stesso oggetto
 - Riferimenti ed *Effetti Collaterali*
- Riferimenti e passaggio dei parametri per valore
- Metodi che restituiscono riferimenti
- Riferimento *null* e **NullPointerException**
- Campo d'Azione delle variabili e *Shadowing*
- La parola chiave **this**
- Convenzioni di Stile

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)

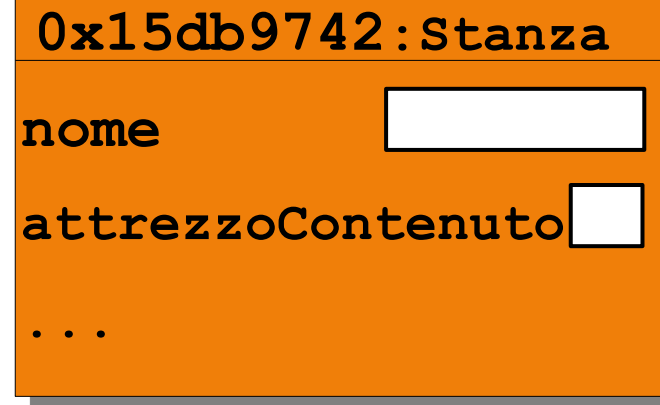
- Una *stessa* variabile può contenere, in momenti diversi, riferimenti ad oggetti distinti dello stesso tipo. Ad esempio:

```
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  
    }  
}
```

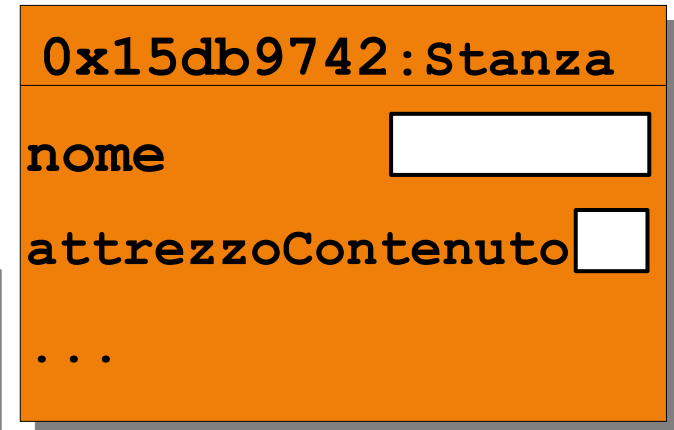
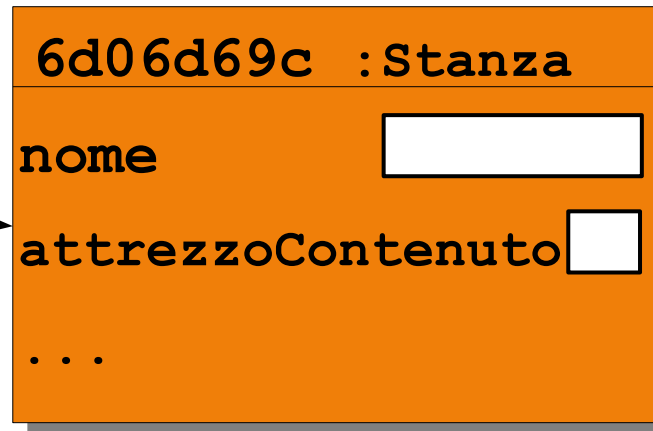
n11 ☐



Riferimenti ad Oggetti (5)

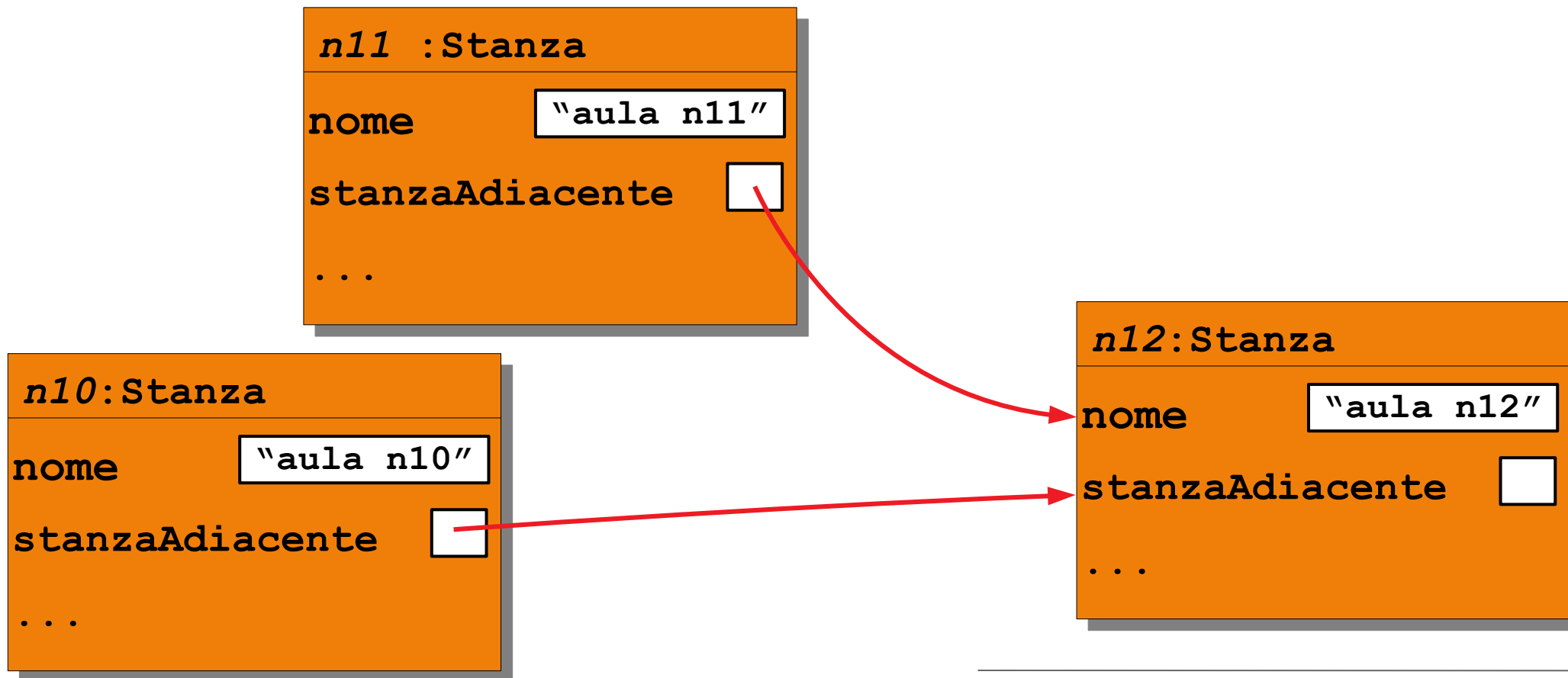
```
public class MainRiferimenti {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
        System.out.println(n11); // stampa Stanza@15db9742  
  
        n11 = new Stanza();  
        System.out.println(n11); // stampa Stanza@6d06d69c  
    }  
}
```

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:

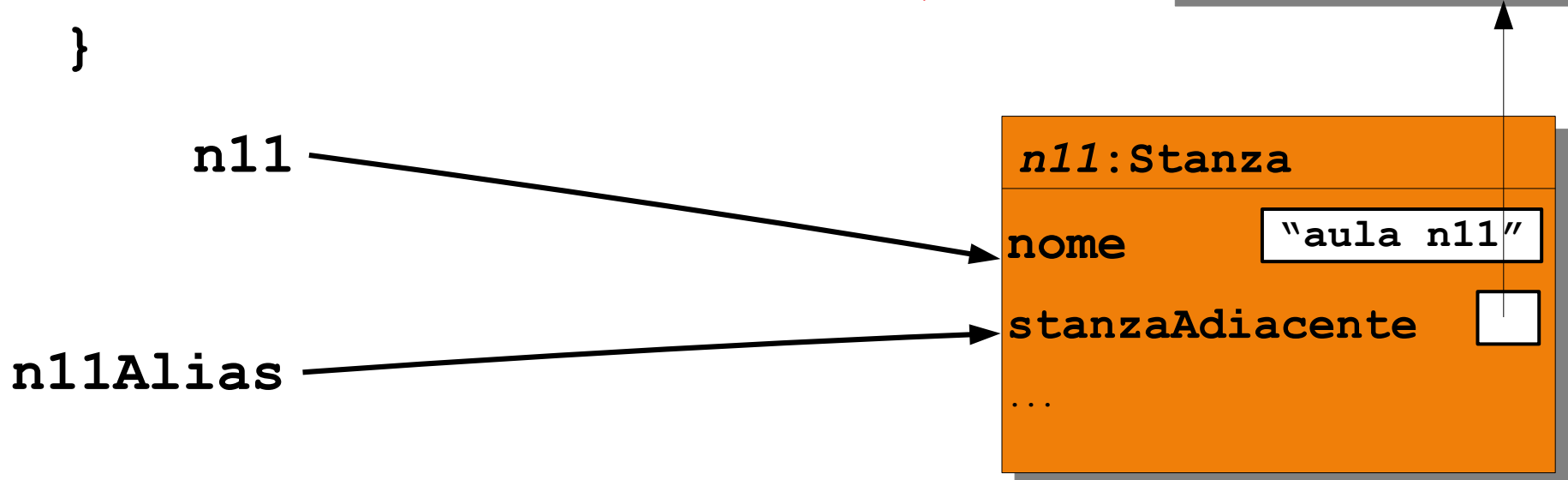
```
public class MainStanzeRiferimenti {  
    public static void main(String[] args) {  
        Stanza n12 = new Stanza();  
        n12.setNome("aula n12");  
  
        Stanza n11 = new Stanza();  
        n11.setNome("aula n11");  
        n11.setStanzaAdiacente(n12);  
  
        Stanza n11Alias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

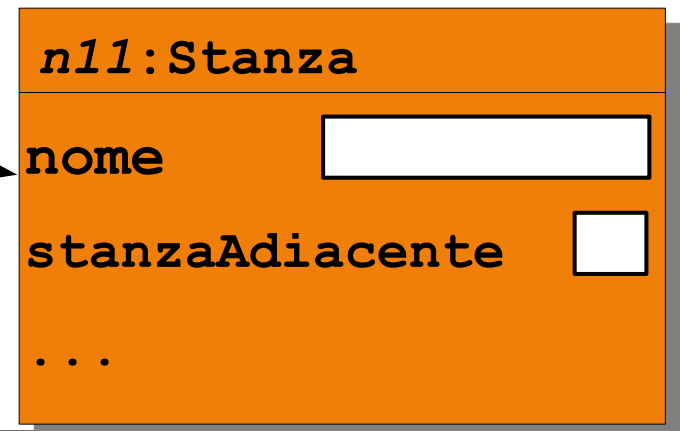
- Qual è l'output del seguente programma?

```
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)

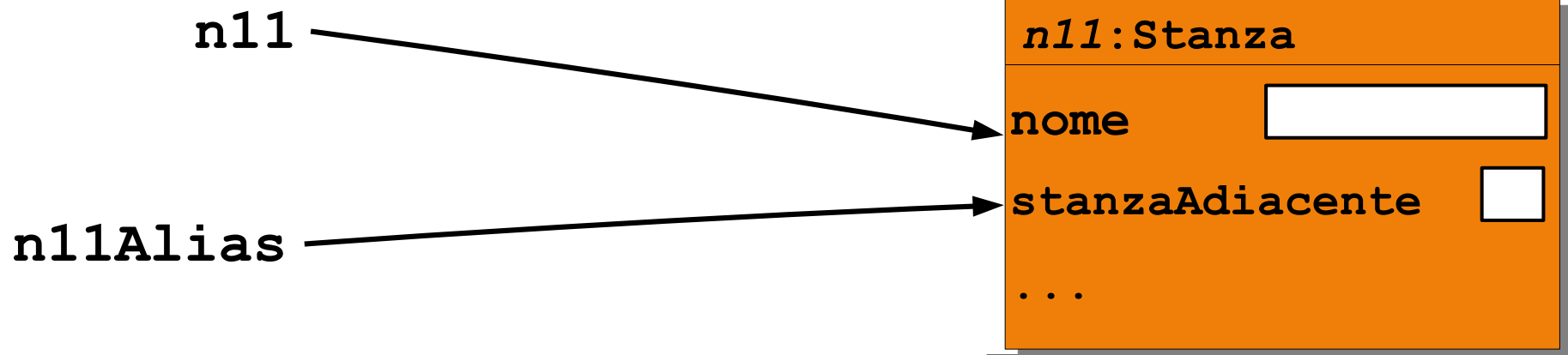
```
public static MainRiferimentiSideEffect {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
        ───────────────────────────────────  
        Stanza n11Alias = n11;  
  
        n11.setNome("N11");  
        n11Alias.setNome("aula N11");  
  
        System.out.println(n11.getNome());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

n11



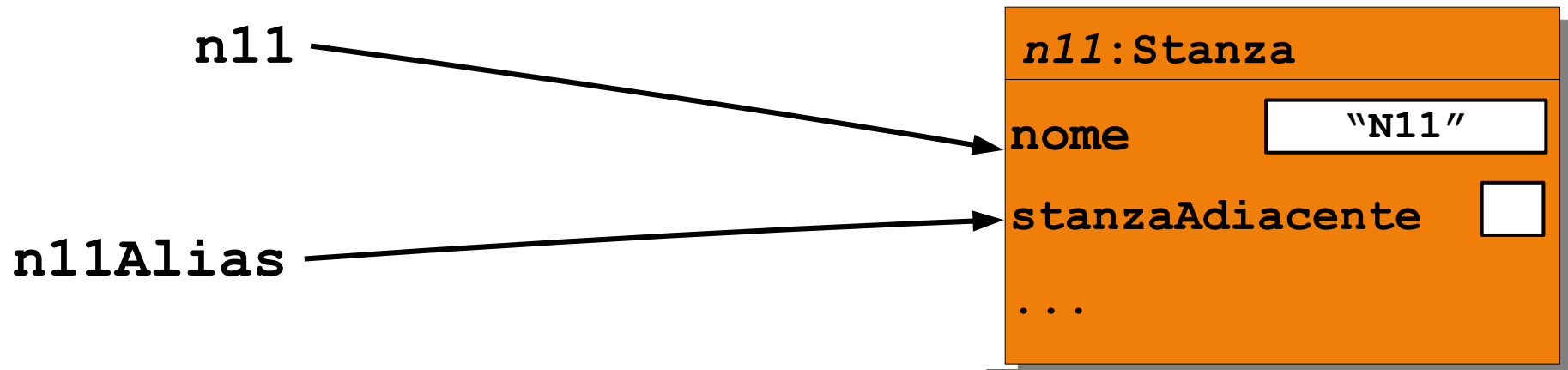
Più Riferimenti & Side-Effect (3)

```
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());  
    }  
}
```



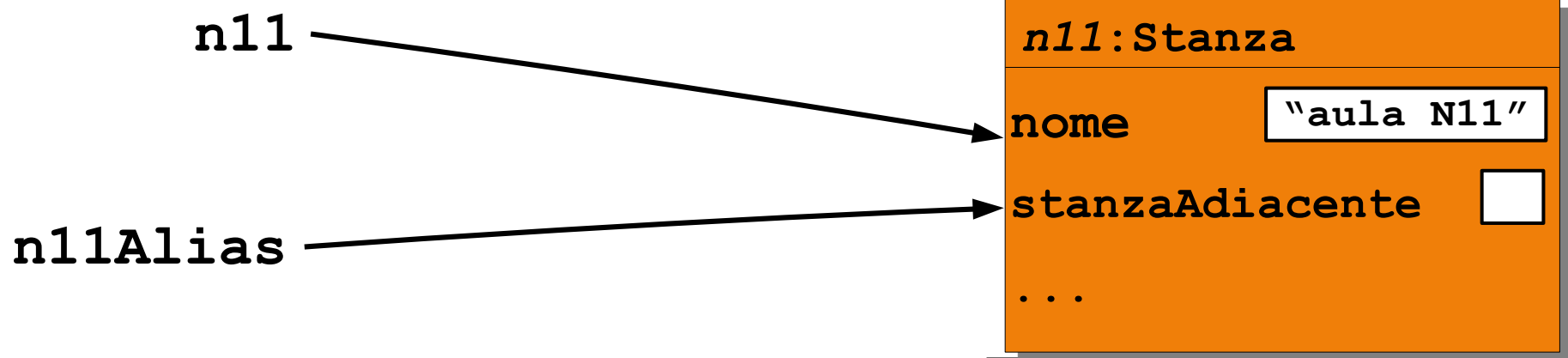
Più Riferimenti & Side-Effect (4)

```
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 per Valore (1)

- 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
- Tramite due copie distinte ma identiche dello stesso riferimento si finisce per accedere (e modificare) lo stesso oggetto
- Il cambio di stato operato ad un oggetto all'interno del *metodo invocato* è visibile (come effetto collaterale) anche al livello *metodo invocante* che effettuato la chiamata passando per valore un riferimento all'oggetto
 - Passando per valore riferimenti si possono quindi ottenere effetti simili a quelli che in altri linguaggi si ottengono mediante il cosiddetto *passaggio dei parametri per variabile*

Riferimenti per Valore (2)

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

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

```
Rettangolo *r = malloc(sizeof(Rettangolo));
```

```
setBase(r, 15);
```

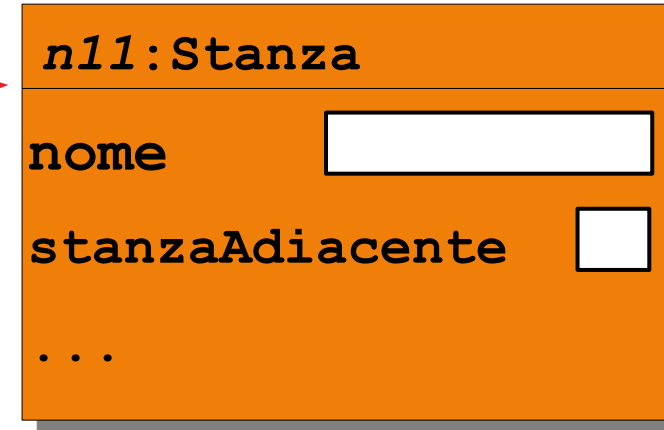
- Similarmente a quanto avviene in C, passando (per valore) il **puntatore** ad un'area di **memoria** allocata con **malloc**
 - è possibile cambiare il contenuto della **memoria** il cui **indirizzo** è fornito come argomento
 - non è possibile cambiare il contenuto della variabile che ospita tale **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 ospita tale **riferimento** al momento dell'invocazione

Passaggio di Riferimenti (1)

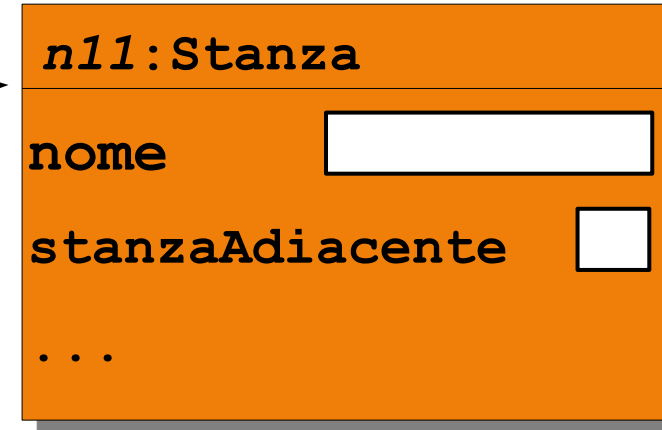
```
public class MainPassRef {  
    public static void main(String[] args) {  
        Stanza n11 = new Stanza();  
        Stanza n12 = new Stanza();  
        n11.setStanzaAdiacente(n12);  
    }  
}
```



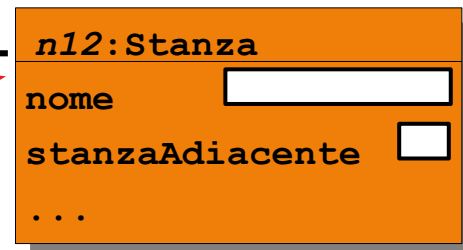
```
public class Stanza {  
    // ...  
    public void setStanzaAdiacente(Stanza stanza) {  
        this.stanzaAdiacente = stanza;  
    }  
}
```

Passaggio di Riferimenti (2)

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

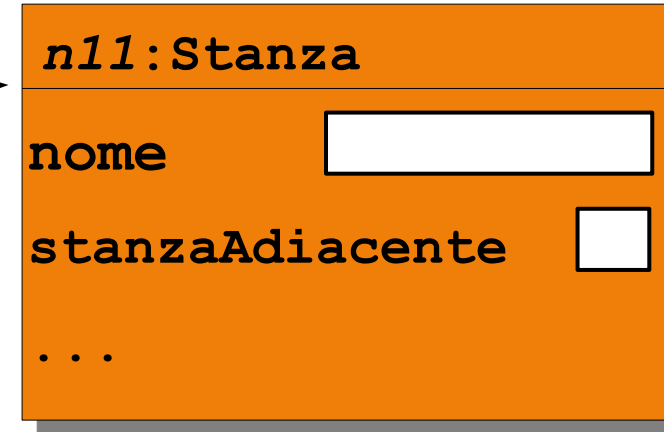


```
public class Stanza {  
    // ...  
  
    public void setStanzaAdiacente(Stanza stanza) {  
        this.stanzaAdiacente = stanza;  
    }  
}
```

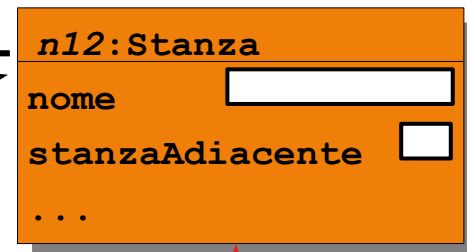


Passaggio di Riferimenti (3)

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

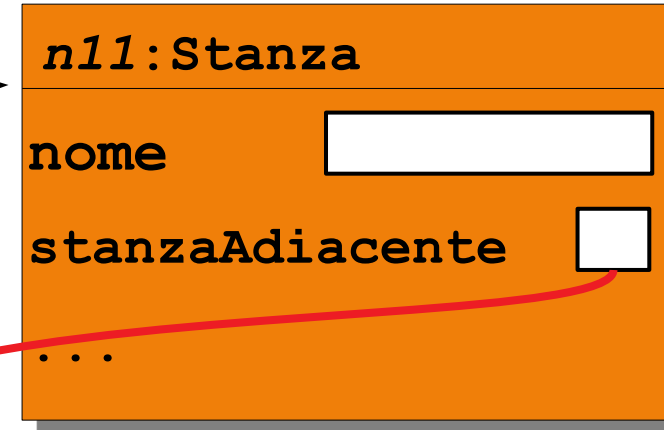


```
public class Stanza {  
    // ...  
    public void setStanzaAdiacente(Stanza stanza) {  
        this.stanzaAdiacente = stanza;  
    }  
}
```

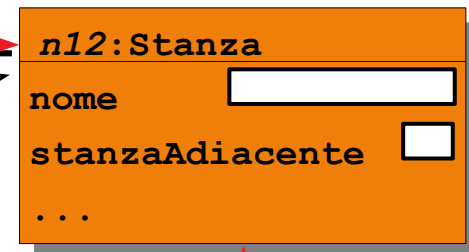


Passaggio di Riferimenti (4)

```
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 risultato dell'effetto collaterale dello spostamento di un punto (il vertice) di cui conservava un riferimento
- E' preferibile dotare la classe **Rettangolo** di un apposito metodo **trasla()** la sua implementazione può fare affidamento ad un metodo (anche con lo stesso nome) di **Punto**
- Gli effetti collaterali risultano difficili da tracciare

Riferimento *Null*

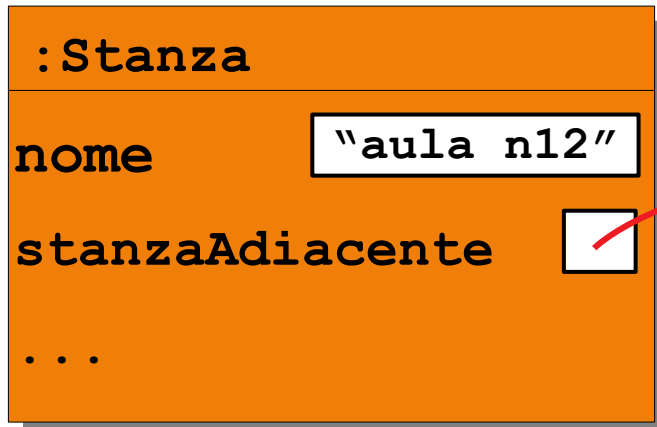
- In Java esiste un solo letterale di tipo riferimento ad oggetto: `null`
- Un valore speciale e distinto da tutti gli altri valori, il riferimento *null*
- Indica l'assenza di un reale riferimento ad un oggetto esistente
 - N.B. In Java non esiste alcuna relazione particolare tra il valore `null` e `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
 - 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*
- La parola chiave **`this`**
 - Conserva un riferimento all'oggetto sul quale il metodo in corso di esecuzione 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 gli altri due metodi della classe `Punto`: `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 consigliabile usare `this` per aumentare la leggibilità
- Ad esempio 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:

- `Studiante.java`
- `Tesi.java`
- `Sommatore.java`