

# Programmazione Orientata agli Oggetti

---

Overloading, Costruttori,  
Stringhe e Array

# Sommario

- Overloading
- Costruttore primario e costruttori secondari
- La classe **String**
  - Il metodo `toString()`
- Diagramma degli oggetti
- Array
- Costanti
- Variabili e metodi di classe

# Overloading (1)

- Caratteristica che permette ad un linguaggio di programmazione di definire molteplici metodi/funzioni/procedure con lo stesso nome
- Una classe Java può ospitare due o più *versioni* dello stesso metodo (ovvero, con lo stesso nome)
- Tale metodo si dice *sovraccarico*
- Le versioni sovraccariche dello stesso metodo devono comunque risultare distinguibili
- Devono differire per la lista di parametri formali
  - per numero di parametri, e/o
  - per il tipo di uno o più parametri, e/o
  - per l'ordine dei parametri

# Overloading (2)

- La classe **Sommatore** può avere più metodi di stesso nome per sommare due interi o tre interi

```
public class Sommatore {  
    public int add(int a, int b) { return a + b; }  
    public int add(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```

- Il tipo e l'ordine dei parametri è significativo:

```
public double add(int a, double b) {  
    return a + b;  
}
```

... è un metodo distinto da:

```
public double add(double a, int b) {  
    return a + b;  
}
```

# Overloading: in C?

- Il linguaggio C non supporta l'overloading
  - Non si possono definire due funzioni con lo stesso nome
- A conferma che questa possibilità offerta dal linguaggio Java è avvertita come una viva esigenza da molti programmatori, basta osservare la convenzione adottata da molti programmatori C per gestire situazioni simili, ovvero:
  - `double add_id(int a, double b)`
  - `double add_di(double a, int b)`
- ✓ I nomi sono resi sintatticamente diversi, con un prefisso comune che rende evidente l'esistenza di una radice comune

# Risoluzione Chiamate in Overloading

- Per ciascuna invocazione di metodo (sovraccarico) si confrontano:
  - tutte le *signature* del metodo, ed in particolare
    - tipo e numero dei parametri formali
  - con la lista dei parametri attuali
- In presenza di una lista di parametri formali in perfetto accordo con la signature di un metodo (per numero e tipo di parametri), la scelta è semplice:
  - Si utilizza la versione di un metodo sovraccarico che segue fedelmente il numero ed il tipo dei parametri attuali

# Chi Risolve le Chiamate Sovraccariche?

- Quando viene operata la scelta?
  - a tempo statico (durante la *compilazione*)  
*oppure*
  - a tempo dinamico (durante l'*esecuzione*)
- E' il **compilatore** a decidere quale versione di un metodo sovraccarico invocare
  - Già durante la compilazione (a tempo statico)
  - La scelta è definitiva, scritta nei **.class** generati
  - Operata esclusivamente sulla base dell'analisi dei tipi della lista di parametri attuali (nell'invocazione di metodo) effettuata durante la compilazione

# Overloading

```
public class ProvaOverloading {  
    public void metodo(int a) {  
        System.out.println("parametro int");  
    }  
    public void metodo(double a) {  
        System.out.println("parametro double");  
    }  
}
```

```
ProvaOverloading prova = new ProvaOverloading();  
prova.metodo(3);      // Stampa "parametro intero"  
prova.metodo(3.0d);  // Stampa "parametro double"
```



# Overloading e Tipo Restituito

- Versioni sovraccariche dello stesso metodo possono differire anche per il tipo del valore restituito:

```
int add(int a, int b);
```

```
double add(double a, int b);
```

- NON è però possibile distinguere due metodi con gli stessi parametri formali (stesso numero, tipo e ordine dei parametri) usando solamente il tipo di ritorno

```
int f(int a, int b);    // Non Compila
```

```
double f(int a, int b); // Non Compila
```

- ✓ Non compila: il compilatore non sarebbe in grado di capire quale versione si dovrebbe invocare (in generale)...

Si usa anche dire che

*il tipo restituito non fa parte della segnatura di un metodo*

# Esempio (con Eclipse)

- All'interno della classe `Rettangolo` scrivere i metodi
  - `void scala(int fattore)`  
deve modificare l'oggetto `Rettangolo` di modo che  
`base = base * fattore`  
`altezza = altezza * fattore`
  - `void scala(int fattoreBase, fattoreAltezza)`  
deve modifica l'oggetto `Rettangolo` di modo che  
`base = base * fattoreBase`  
`altezza = altezza * fattoreAltezza`

# Esempio (2)

```
public class Rettangolo {  
    private int base;  
    private int altezza;  
    // ...  
    public void scala(float fattore) {  
        this.scala(fattore, fattore);  
    }  
    public void scala(float fattoreBase,  
                      float fattoreAltezza) {  
        this.base *= fattoreBase;  
        this.altezza *= fattoreAltezza;  
    }  
}
```

# Esempio (3)

- Notare come il metodo  
`scala(float fattore)`

faccia uso del metodo

`scala(float fattoreBase, float fattoreAltezza)`

...per evitare la duplicazione del codice (>>)

# Risoluzione Metodi Sovraccarichi (1)

- In assenza di una corrispondenza perfetta tra
  - lista dei parametri attuali di una invocazione
  - segnatura di un metodo

il compilatore applica un algoritmo di risoluzione che prima di fermare la compilazione cerca di capire se semplici conversioni di tipo permettono la chiamata

- Ad es. `int` → `float`
- L'algoritmo è intriso di dettagli per coprire l'ampia casistica; in pratica basta quasi sempre limitarsi a ricordare che:
  - sono ammesse semplici promozioni di tipo se necessarie a rendere un'invocazione confacente con una delle signature disponibili
  - sono preferite le promozioni più “conservative”, ovvero quelle al tipo immediatamente più grande

# Risoluzione Metodi Sovraccarichi (2)

- Più nel dettaglio:
  - Argomento di tipo intero
    - se esiste un metodo che prende come parametro formale `int`, allora viene usato quel metodo
    - altrimenti viene promosso al tipo di dato più piccolo tra quelli disponibili (ma “capaci di contenere” un `int`):  
`long`, `float`, `double`
  - Argomento in virgola mobile
    - Si cerca il metodo che ha come parametro formale un `double`
  - Argomento di tipo carattere
    - Se non trova una corrispondenza con `char`, si prova la promozione a `int`

# Risoluzione Metodi: Esempio

Motivo di confusione è la promozione *implicita* di tipo per i tipi primitivi. Ad esempio:

```
public class Prova {  
    void f(long i)    { System.out.println("long");    }  
    void f(float i)   { System.out.println("float");   }  
    void f(double i) { System.out.println("double"); }  
    public static void main(String[] args) {  
        f(5); // ???  
    }  
}
```

```
public class Prova {  
    void f(long i)    { System.out.println("long");    }  
    void f(float f)   { System.out.println("float");   }  
    void f(double i) { System.out.println("double"); }  
    public static void main(String[] args) {  
        f(5); // ???  
        f(5f); // ???  
    }  
}
```

# Con Riferimenti: Cosa Stampa?

```
public class Boo {  
  
    void f(String n) {  
        System.out.println("stringa");  
    }  
  
    void f(int n) {  
        System.out.println("intero");  
    }  
  
    public static void main(String[] args) {  
        Boo b = new Boo();  
        String s = new String("pppp");  
        int i = 0;  
        b.f(s);  
        b.f(i);  
    }  
}
```



# Overloading di Operatori

- L'overloading è una caratteristica comoda ma usata in maniera decisamente controllata in Java
- Anche l'operatore `+` è sovraccarico: così come già accade in C, risulta più agevole la manipolazione di tipi numerici
  - Somma interi (`int`, `long`)
  - Somma numeri in virgola mobile (`float`, `double`)
- Invece specifica del linguaggio Java è la sua applicazione ad una classe con un supporto particolare: **String**
  - Concatenazione di stringhe; Ad es.:
  - `System.out.println("Ciao"+" "+"Mondo");`
- In C++ (ed in Scala) c'è anche la possibilità di sovraccaricare tutti gli altri operatori, in Java (fortunatamente) NO
  - per il pessimo rapporto costi/benefici

# Costruttori ed Overloading

- Anche i costruttori possono essere sovraccarichi
- Ad esempio la classe `Rettangolo` potrebbe avere i seguenti costruttori
  - Un costruttore *no-args*. Quindi
    - `base = 0; altezza = 0, vertice = (0, 0)`
  - Un costruttore che ha come parametri base e altezza. Quindi:
    - `vertice = (0, 0)`
  - Un costruttore generico
    - `vertice`
    - `base`
    - `altezza`

# Esempio (*con Eclipse*)

- Realizzare i costruttori della classe **Rettangolo** appena descritti

# Esempio (2)

```
public class Rettangolo {  
    private int altezza;  
    private int base;  
    private Punto vertice;  
    public Rettangolo(Punto vert, int base, int altezza) {  
        this.vertice = vert;  
        this.base = base;  
        this.altezza = altezza;  
    }  
    public Rettangolo(int base, int altezza) {  
        this.vertice = new Punto(0, 0);  
        this.base = base;  
        this.altezza = altezza;  
    }  
    public Rettangolo() {  
        this.vertice = new Punto(0, 0);  
        this.base = 0;  
        this.altezza = 0;  
    }  
}
```

# Esempio (3)

```
public class MainOverloading {  
    public static void main(String[] args) {  
        // base 0, altezza 0, vertice in (0, 0)  
        Rettangolo r1 = new Rettangolo();  
  
        // base 3, altezza 5, vertice in (0, 0)  
        Rettangolo r2 = new Rettangolo(3, 5);  
  
        Punto vertice = new Punto(4, 9);  
        Rettangolo r3 = new Rettangolo(vertice, 3, 5);  
    }  
}
```

# Costruttore “Primario”

- Costruttori definiti ripetendo molto codice, come appena fatto sono... *sconsigliabili!*
- Le ripetizioni nel codice causano problemi (>>)
- Meglio eleggere un costruttore al ruolo di “*primario*”, il più generico possibile
- Eccolo per la classe **Rettangolo**:

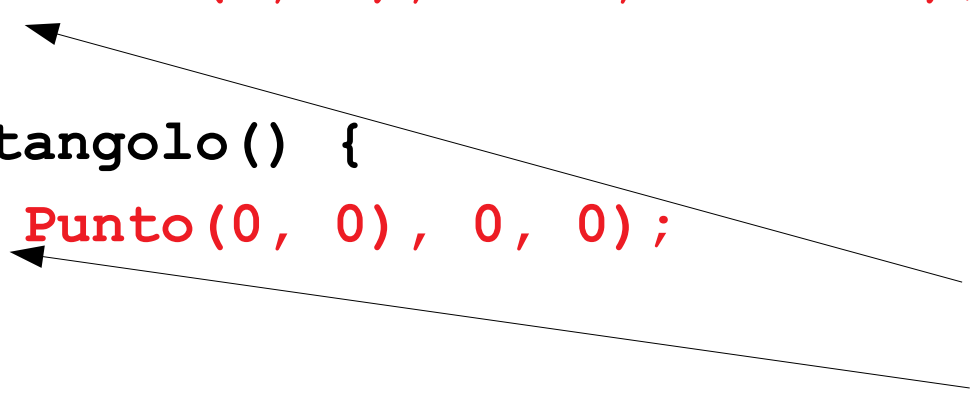
```
public Rettangolo(Punto vertice, int base, int altezza) {  
    this.vertice = vertice;  
    this.base = base;  
    this.altezza = altezza;  
}
```

# Costruttori “Secondari”

- A questo punto, gli altri costruttori (“*secondari*”) possono affidarsi a quello più generico
- Per invocare un costruttore da un costruttore:  
**this(<lista di argomenti>)**
  - L'invocazione di un altro costruttore *deve essere la prima istruzione* nel corpo del costruttore *secondario*
    - Altrimenti: errore di compilazione
- I costruttori secondari finiscono per fissare dei valori predefiniti per tutti i parametri che non ricevono esplicitamente
- In Scala esiste una sintassi apposita per distinguere costruttori *primari* dai *secondari*, in Java NO

# Costruttori Chiamati dai Costruttori

```
public class Rettangolo {  
    private int altezza;  
    private int base;  
    private Punto vertice;  
    public Rettangolo(Punto vert, int base, int altezza) {  
        this.vertice = vert;  
        this.base = base;  
        this.altezza = altezza;  
    }  
    public Rettangolo(int base, int altezza) {  
        this(new Punto(0, 0), base, altezza);  
    }  
    public Rettangolo() {  
        this(new Punto(0, 0), 0, 0);  
    }  
}
```



Crea un nuovo oggetto **Punto** ed invoca un altro costruttore entro un'unica (la prima) riga di codice



# La Classe String (1)

- In Java esiste la classe **String** per rappresentare sequenze di caratteri *immutabili*
  - Le stringhe sono oggetti, **String** è la classe a cui appartengono le sue istanze
- Una variabile dichiarata di tipo **String** contiene quindi un *riferimento* ad un oggetto istanza della classe **String**
- `String favorita = new String("Sono la favorita!");`
- Pur essendo una classe come tutte le altre, spesso si ha la tentazione di pensare che non lo sia affatto...
- Il motivo è che ha un supporto *molto* particolare sia nel linguaggio che da parte del compilatore

# La Classe String (2)

- Per rendere il linguaggio più semplice ed attraente per nuovi sviluppatori, all'epoca si decisero trattamenti decisamente “di favore” per questa classe
- Prima di Java 5 era (>>)
  - l'unica classe che possiede dei letterali appositi
  - l'unico tipo di oggetto che si può creare senza fare una **new** esplicita
- **String favorita = "Sono la favorita!";**  
equivale a
- **String favorita = new String("Sono la favorita!");**
- L'inserimento dell'invocazione della **new** è operata direttamente dal compilatore
- Anche il fatto che l'operatore **+** sia sovraccarico per gestire la concatenazione conferma il trattamento di favore...
- Sicuramente tutto questo “opacizza” il codice (>>)

# Equivalenza di Stringhe (1)

- Dato che le stringhe sono oggetti *veri* e propri, l'equivalenza deve essere valutata mediante il metodo appositamente previsto dalla classe `equals()`  
Infatti:

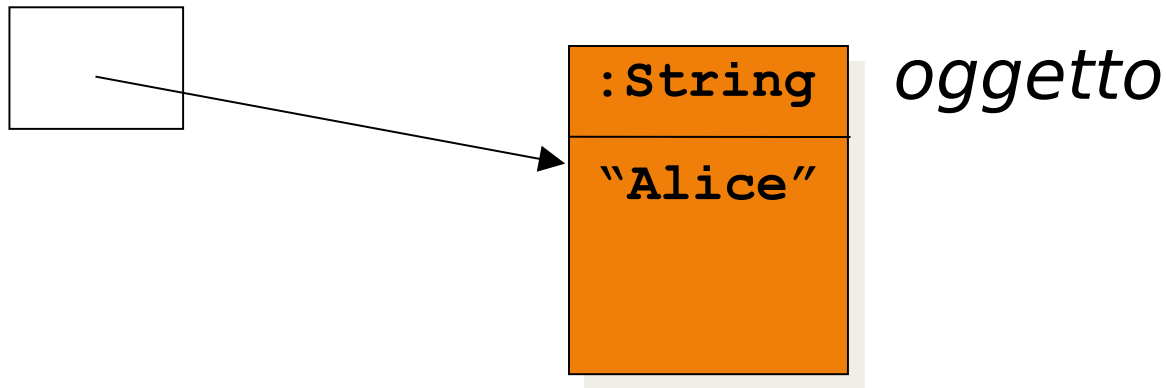
- `String nome = new String("Alice");`
- `String omonimo = new String("Alice");`

```
System.out.println(nome == omonimo);  
// Stampa false
```

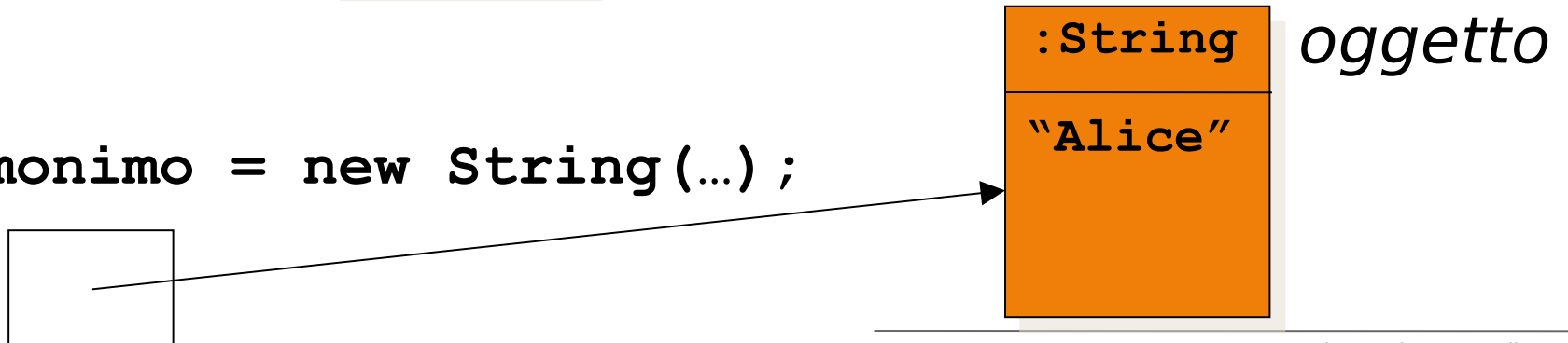
# Equivalenza di Stringhe (2)

- Si sta in realtà verificando che i *referimenti* risultino identici
  - *falso*: `nome` e `omonimo` contengono riferimenti diversi, ovvero restituiti da due distinte invocazioni della `new`

```
String nome = new String(...);
```



```
String omonimo = new String(...);
```



# Equivalenza di Stringhe (3)

- Invece, utilizzando:

```
String nome = new String("Alice");  
String omonimo = new String("Alice");  
  
System.out.println(nome.equals(omonimo));  
// stampa true
```

il metodo `equals()` controllerà l'equivalenza della sequenza di caratteri che compone le stringhe, carattere per carattere

# Immutabilità della Classe String

- Gli oggetti `String` usati per rappresentare stringhe in Java sono *immutabili*
  - Non è possibile modificare i caratteri all'interno di una stringa una volta creata
  - Non si possono aggiornare: bisogna sempre crearne di nuove per memorizzare la modifica
- Ad esempio per concatenare due stringhe:

```
String s1 = "ciao "; // = new String("ciao ");  
String s2 = "mondo"; // = new String ("mondo");  
s1 = s1 + s2; // → Si sta creando un nuovo oggetto  
// String e si sta sovrascrivendo il vecchio  
riferimento
```

```
System.out.println(s1); // ciao mondo
```

# Metodi della Classe String (1)

- `String s = "POO";`
  - Lunghezza della stringa  
`s.length()` ; restituisce **3**
  - Ottenere un carattere in una certa posizione  
`s.charAt(0)` ; restituisce '**P**'
    - Indicizzazione base 0
    - come per gli array il primo carattere ha indice 0

# Metodi della Classe String (2)

- `String s = "una stringa";`
  - Indice di un carattere  
`s.indexOf('s');` restituisce 4
  - Indice di una stringa  
`s.indexOf("ring");` restituisce 6
- Il metodo `indexOf()` è sovraccarico
- Restituisce -1 se non trova il carattere o la stringa cercata  
`s.indexOf('z');` restituisce -1



# Metodi della Classe String (3)

- `String s = "una stringa";`
  - Rimpiazzare caratteri
- `String s2 = s.replace("stringa", "stringa lunga");`

```
System.out.println(s); // Stampa 'una stringa'
```

```
System.out.println(s2); // Stampa 'una stringa lunga'
```

- Le stringhe sono immutabili: in realtà si crea un nuovo oggetto `String` il cui riferimento finisce in `s2`
- *molti e molti altri metodi ancora....*

<https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

# Stampare Descrizioni di Oggetti

- Se si esegue `println()` su un riferimento ad un oggetto verrà stampato il valore di tale riferimento

```
// un attrezzo di nome spada e peso 7 kg
```

- `Attrezzo spada = new Attrezzo("spada", 7);`

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

- Stampa, ad es.: `Attrezzo@70dea4e`
  - Non descrive affatto il contenuto e lo stato dell'oggetto, ma il riferimento

# Il Metodo `toString()` (1)

- È possibile cambiare questo comportamento implementando di segnatura:

```
public String toString()
```

per tutte le classi in cui si vuole specificare come trasformare gli oggetti in stringhe

- Il metodo `toString()` restituisce la rappresentazione dell'oggetto sotto forma di stringa
- Addirittura fondamentale per rendere agevole il *debugging* ed il *tracing*

# Il Metodo toString() (2)

- Ad esempio in **Attrezzo**

```
public class Attrezzo {  
    private String nome;  
    private int peso;  
    public Attrezzo(String nome, int peso) {  
        this.nome = nome; this.peso = peso;  
    }  
    // getter  
    public String toString() {  
        return "Attrezzo di nome " + this.getNome() +  
            ". Peso: " + this.getPeso();  
    }  
}
```

# Il Metodo toString() (3)

```
public class MainToString {  
    public static void main(String[] args) {  
        • Attrezzo spada = new Attrezzo("spada", 7);  
  
        System.out.println(spada);  
    }  
}
```

- Stampa

Attrezzo di nome spada. Peso: 7

# Il Metodo toString() (4)

- Stesso comportamento concatenando un riferimento ad un oggetto con un letterale stringa:

```
public class MainToString {  
    public static void main(String[] args) {  
        Attrezzo spada = new Attrezzo("spada", 7);  
        String descr = "attrezzo posseduto: " + spada;  
        System.out.println(descr);  
    }  
}
```

Stampa:

attrezzo posseduto: Attrezzo di nome spada. Peso: 7

# `println()` & `toString()`

- Il metodo `toString()` (come `equals()`), anche è presente in tutti gli oggetti che creiamo (>>)
- Tuttavia:
  - Se non esplicitamente implementato stampa [un numero che dipende dal]l'indirizzo di memoria dell'oggetto su cui è invocato
  - L'invocazione del metodo `toString()`, è inserita direttamente dal compilatore, nelle istruzioni di stampa (anche se non esplicitamente richiesta!):
    - `System.out.println(oggRef)`  
stampa il risultato di `oggRef.toString()`

# Diagramma delle Classi

- Diagrammi UML
  - Ne esistono diversi tipi
- Sono un utile e comodo strumento a supporto (e non in sostituzione) della comunicazione
  - Tempo dinamico: abbiamo già utilizzato rappresentazioni diagrammatiche degli oggetti
  - Tempo statico: diagrammi delle classi
- Il diagramma delle classi illustra le caratteristiche principali (variabili di istanza, costruttori e metodi) delle classi che compongono la applicazione
- L'enfasi è sulla relazione tra le classi, quindi sugli aspetti “statici”

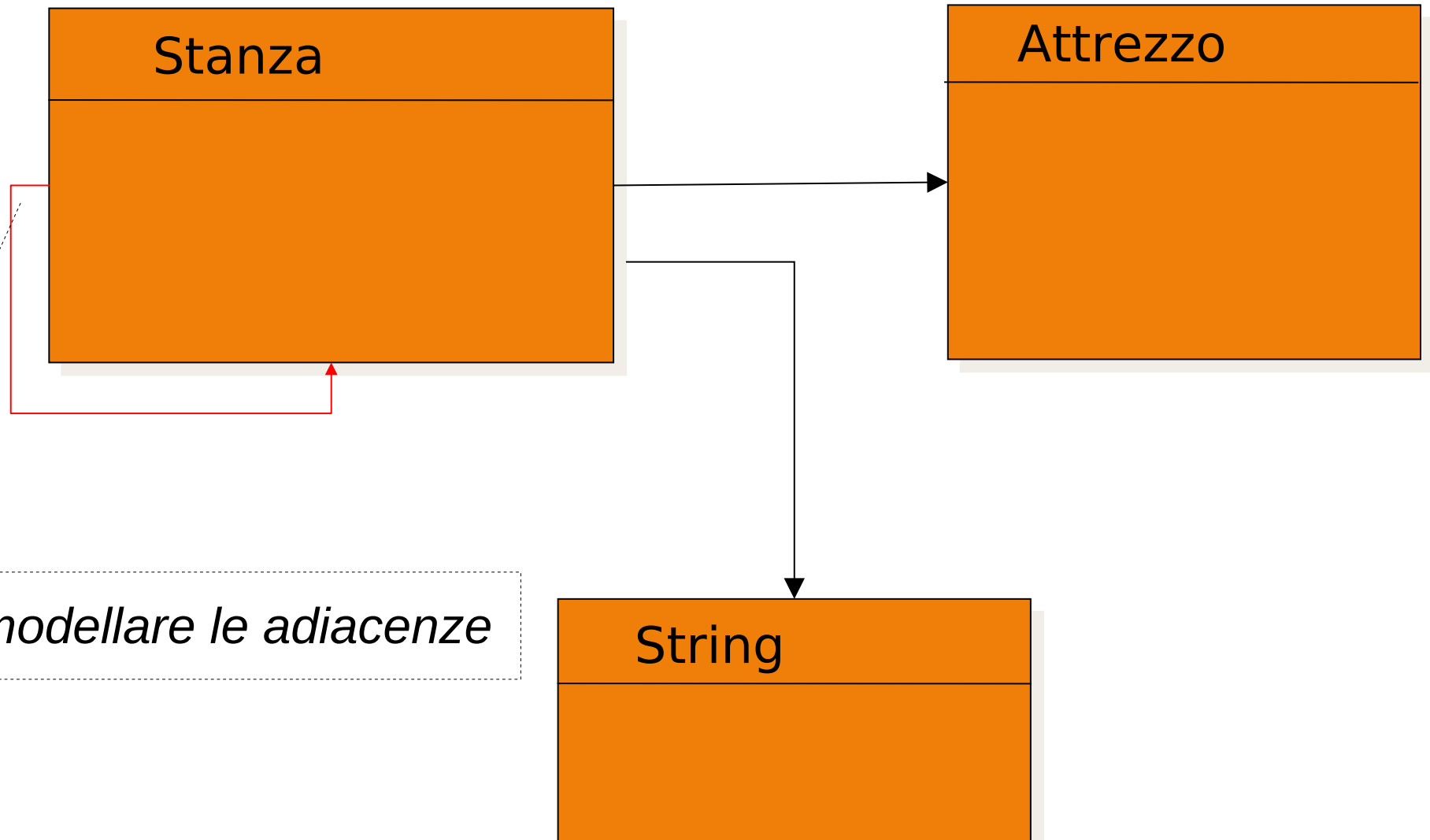


# Esempio

- Supponiamo

```
public class Stanza {  
    private String nome;  
    private Attrezzo attrezzo;  
  
    public Stanza(String nome) {  
        this.nome = nome;  
    }  
  
    public void setAttrezzo(Attrezzo attrezzo) {  
        this.attrezzo = attrezzo;  
    }  
    ... altri metodi  
}
```

# Diagramma delle Classi: Esempio



*Per modellare le adiacenze*

# Diagramma degli Oggetti (1)

- Per avere un'idea della evoluzione di un programma è utile rappresentare lo stato delle istanze: a tal fine usiamo una rappresentazione grafica, chiamata **diagramma degli oggetti**
- Il diagramma degli oggetti mostra gli oggetti istanziati in memoria durante l'esecuzione dell'applicazione
- L'enfasi è sullo stato interno degli oggetti e sugli aspetti dinamici
  - ogni oggetto ha un indirizzo di memoria
  - le variabili di tipo riferimento ad oggetto memorizzano l'indirizzo dell'oggetto referenziato
  - una rappresentazione grafica efficace basata sull'uso di frecce...

# Esempio

...

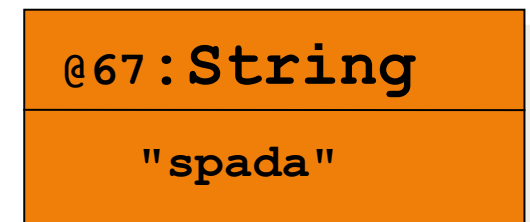
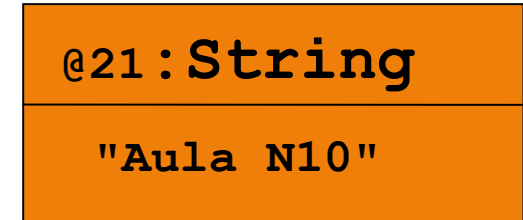
```
public static void main(String[] args) {  
    Attrezzo spada = new Attrezzo("spada", 10);  
    Stanza n10 = new Stanza("Aula N10");  
    n10.addAttrezzo(spada);
```

```
// < ---
```

```
/* disegnare il diagramma degli oggetti  
   in questo punto dell'esecuzione */
```

```
}
```

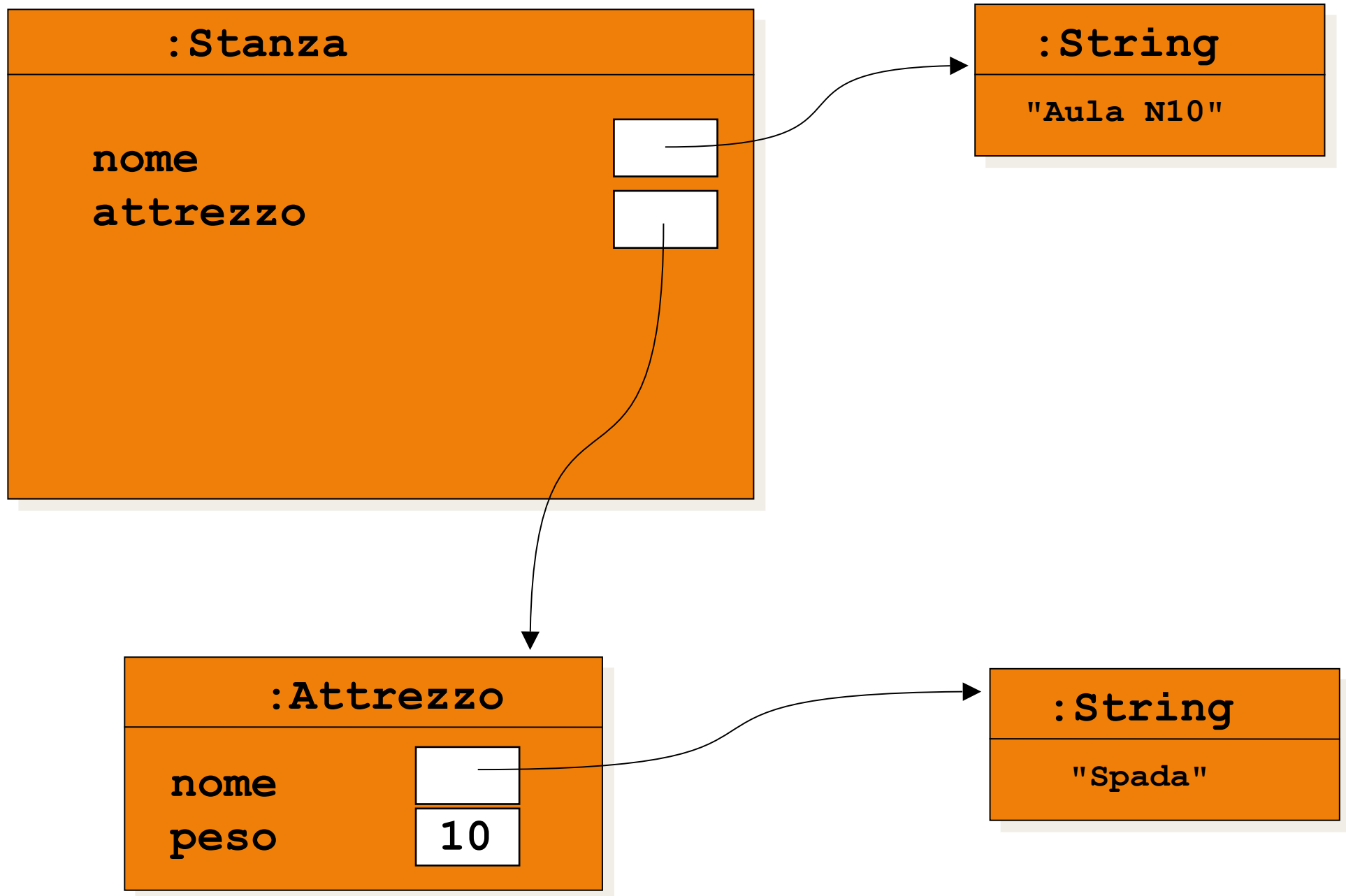
# Diagramma degli Oggetti: Esempio (1)



# Diagramma degli Oggetti (2)

- Una rappresentazione grafica efficace dei valori memorizzati nelle variabili riferimento prevede l'uso di frecce che collegano
  - la variabile riferimento
  - all'oggetto referenziato

# Diagramma degli Oggetti: Esempio (2)



# Tipi Primitivi e Oggetti

- Dal diagramma degli oggetti notiamo che le variabili (di istanza) possono memorizzare
  - tipi primitivi
  - riferimenti ad oggetti



# Tipi Primitivi in Java

<b>boolean</b>	vero ( <b>true</b> ) o falso ( <b>false</b> )
<b>char</b>	caratteri Unicode 2.1 (16-bit)
<b>byte</b>	interi a 8 bit (con segno e in C2)
<b>short</b>	interi a 16 bit (con segno in C2)
<b>int</b>	interi a 32 bit (con segno in C2)
<b>long</b>	interi a 64 bit (con segno in C2)
<b>float</b>	numeri in virgola mobile a 32-bit
<b>double</b>	numeri in virgola mobile a 64-bit

# Tipi Primitivi e Oggetti (1)

## Tipo primitivo

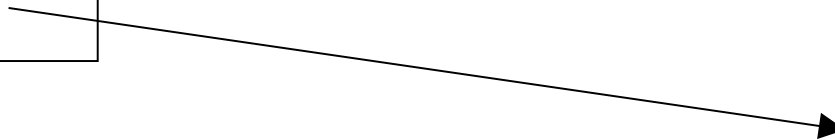
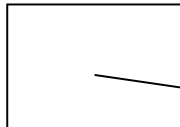
```
int i;
```



---

## Riferimento ad Oggetto

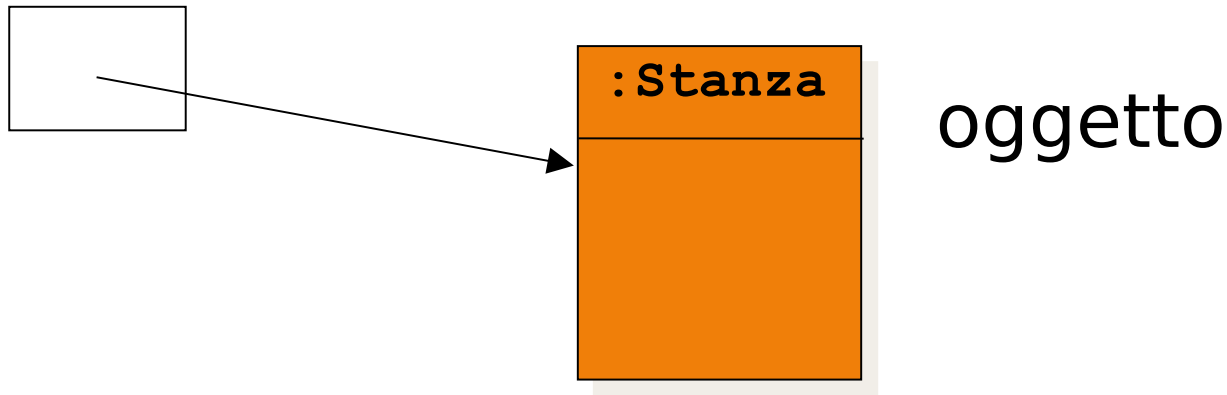
```
Stanza s;
```



Oggetto

# Tipi Primitivi e Oggetti (2)

```
Stanza s = new Stanza (...);
```



- La variabile *s* *non* memorizza un oggetto (nell'esempio, una istanza della classe `Stanza`), ma un *riferimento* all'oggetto

# Tipi Primitivi e Oggetti (3)

```
int i;
```

32

Tipo primitivo

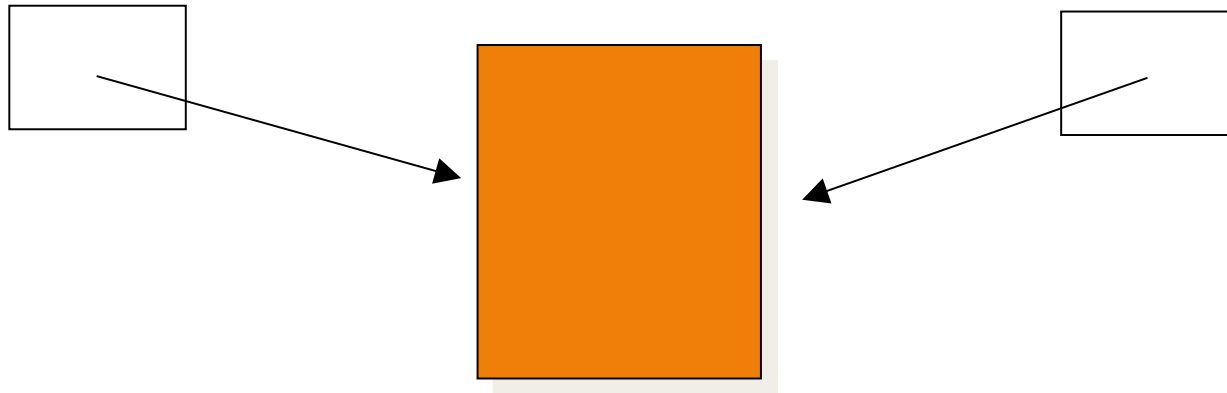
- Nel caso dei tipi primitivi, il valore è memorizzato direttamente nella variabile

# Tipi Primitivi e Oggetti (4)

Oggetti

`MiaClasse a;`

`MiaClasse b;`



---

`a = b;`

Tipi primitivi

`int a;`

32

`int b;`

32

# Tipi Primitivi e Oggetti (5)

```
int i1 = 0;
```

```
int i2 = 5;
```

```
i1 = i2;
```

i1

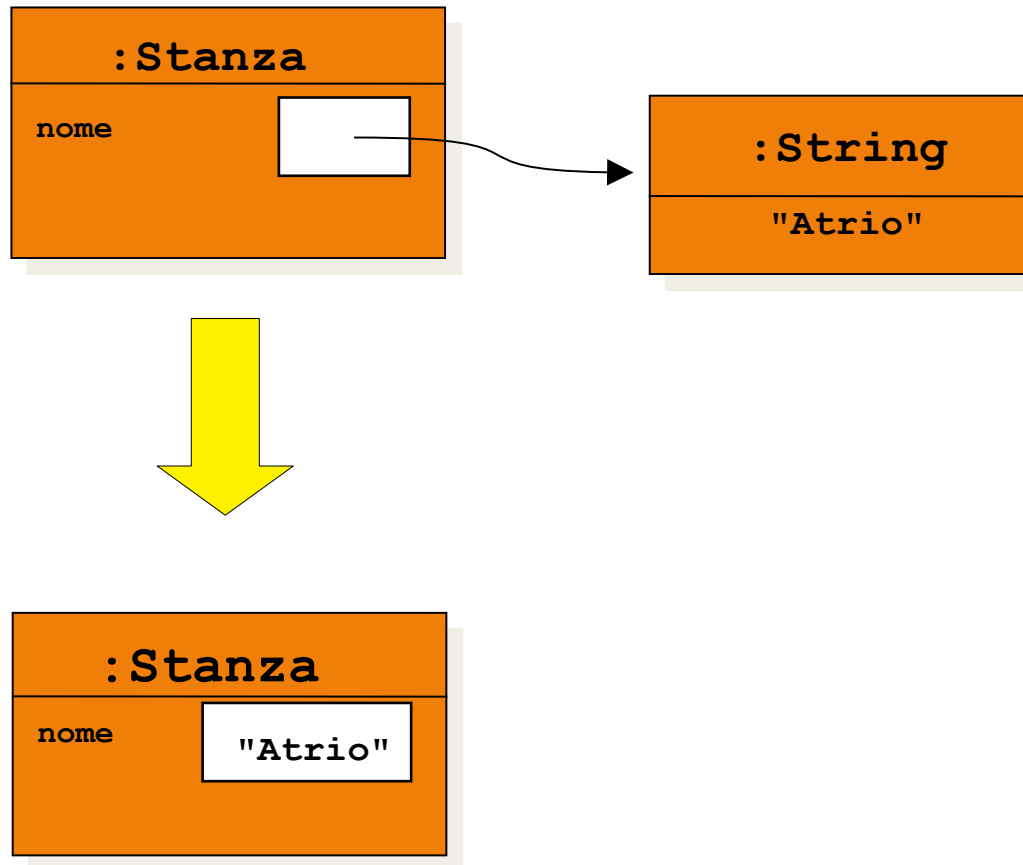
5

i2

5

# String nei Diagrammi ad Oggetti

- Come il compilatore Java, anche noi riserviamo un trattamento di favore agli oggetti istanza della classe `String`



# Tipi Primitivi e Oggetti (6)

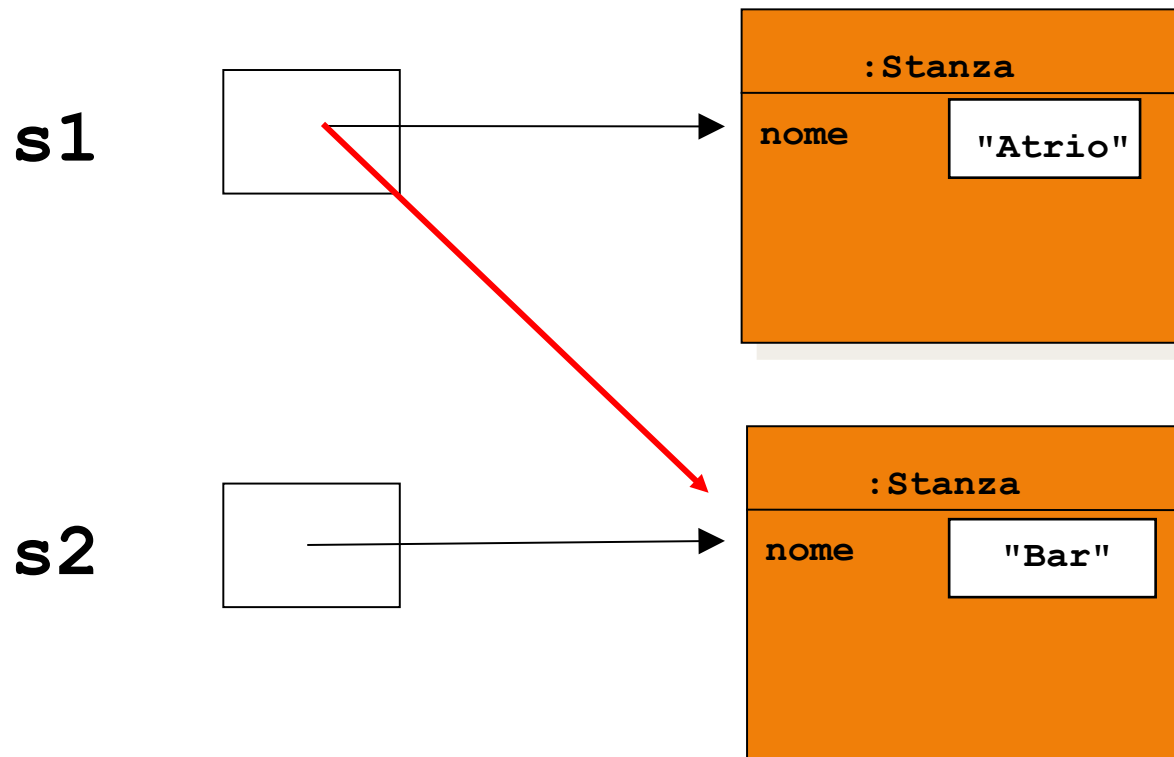
```
Stanza s1;
```

```
s1 = new Stanza("atrio");
```

```
Stanza s2;
```

```
s2 = new Stanza("bar");
```

```
s1 = s2;
```





# Esercizio

- Disegnare il diagramma degli oggetti che rappresenta lo stato degli oggetti referenziati dalle variabili **a**, **b**, **c** del seguente programma al termine della esecuzione della istruzione in linea 3

```
1. Attrezzo a = new Attrezzo("spada", 40);  
2. Attrezzo b = new Attrezzo("scudo", 30);  
3. Attrezzo c = new Attrezzo("lancia", 10);  
4. a = b;
```

# Esercizio

- Disegnare il diagramma degli oggetti che rappresenta lo stato degli oggetti referenziati dalle variabili **a**, **b**, **c** del seguente programma al termine della esecuzione della istruzione in linea **4**

```
1. Attrezzo a = new Attrezzo("spada", 40);  
2. Attrezzo b = new Attrezzo("scudo", 30);  
3. Attrezzo c = new Attrezzo("lancia", 10);  
4. a = b;
```

# Esercizio

- Disegnare il diagramma degli oggetti che rappresenta lo stato del seguente programma al termine della esecuzione della istruzione in linea 5
- Quale valore ha il peso dell'attrezzo che si trova nella stanza referenziata dalla variabile **s** al termine della istruzione 4? E al termine della istruzione 5?

```
1. Attrezzo a = new Attrezzo("spada", 40);  
2. Attrezzo b = new Attrezzo("lancia", 10);  
3. Stanza s = new Stanza("N10");  
4. s.setAttrezzo(a);  
5. a = b;  
6. System.out.println(s.toString()); //quale attrezzo è  
7.                               //presente nella stanza s?
```

# Array (1)

- Un array definisce una struttura di dati che memorizza un insieme di valori dello stesso tipo
- Dichiarazione di una variabile array:  

```
int[] a;
```
- L'oggetto array va creato, specificando il numero di elementi  

```
a = new int[10];
```
- Un array può essere inizializzato esplicitamente al momento della creazione:  

```
int[] a = {21,12,23,34,15,21,7,80,1,-21};
```
- In ogni caso verrà inizializzato con dei valori di default

# Array (2)

- Le sintassi  
`int[] array` e `int array[]`  
sono equivalenti.
- `int[] array` forse più esplicita:
  - già dal prefisso della dichiarazione si capisce che la variabile seguente è un array

# Array (3)

- È possibile accedere a ciascun valore dell'array mediante un indice

```
int[] a;  
int i;  
a = new int[10];  
i = a[4];  
a[6] = 3*i;
```

- Attenzione: gli array in Java (come in C) usano *base-0*: l'indice del primo elemento è 0
- Per avere la dimensione di un array: **.length**
- Scansione degli elementi di un array:

```
for (int i = 0; i < a.length; i++)  
    System.out.println(a[i]);
```

# Array (4)

- A partire da Java 5 è stata introdotta una variante del costrutto `for` (chiamata *for-each*) che consente di scorrere gli elementi di un array (e, come vedremo, anche di altre tipologie di collezioni) senza gestire esplicitamente l'indice di iterazione:

```
int a[];  
a = new int[100];  
for (int elemento : a)  
    System.out.println(elemento);
```

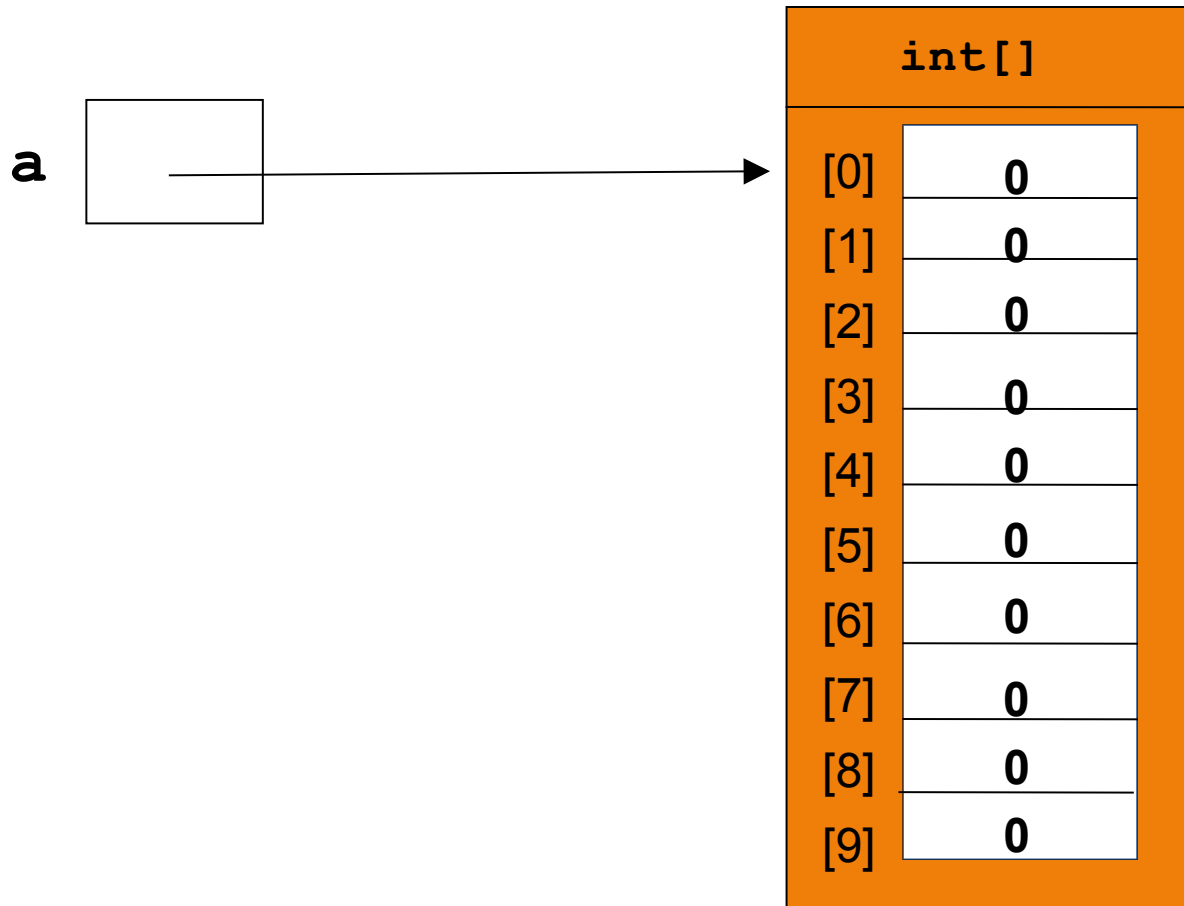
***equivale a:***

```
for (int i=0; i<a.length; i++) {  
    int elemento = a[i];  
    System.out.println(elemento);  
}
```

# Array: Diagramma degli Oggetti

```
int[] a;
```

```
a = new int[10];
```



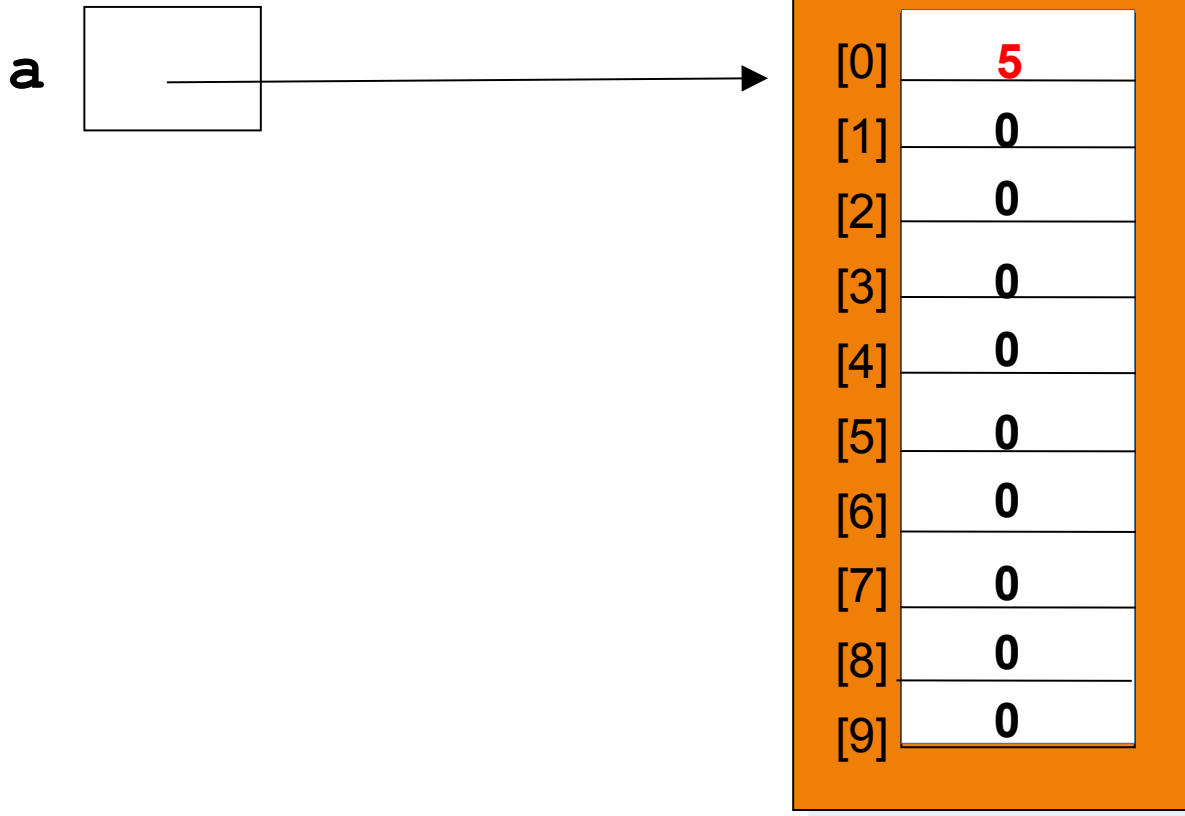


# Array: Diagramma degli Oggetti

```
int[] a;
```

```
a = new int[10];
```

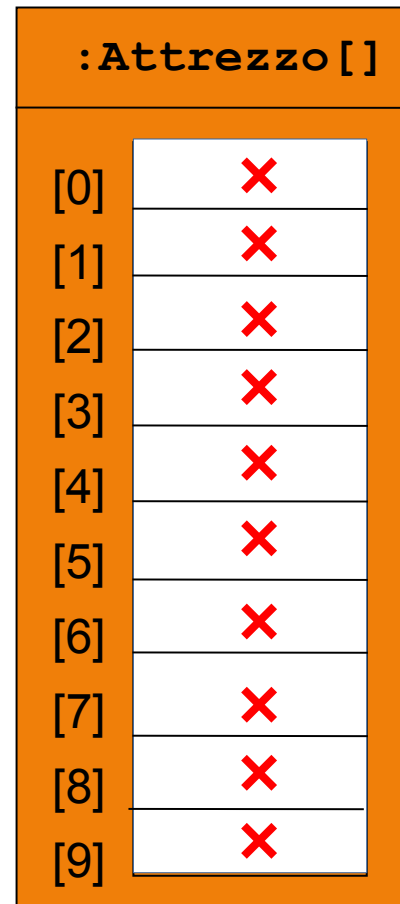
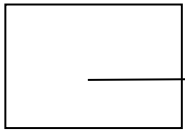
```
a[0] = 5;
```



# Array: Diagramma degli Oggetti

```
Attrezzo attrezzi[];  
attrezzi = new Attrezzo[10];
```

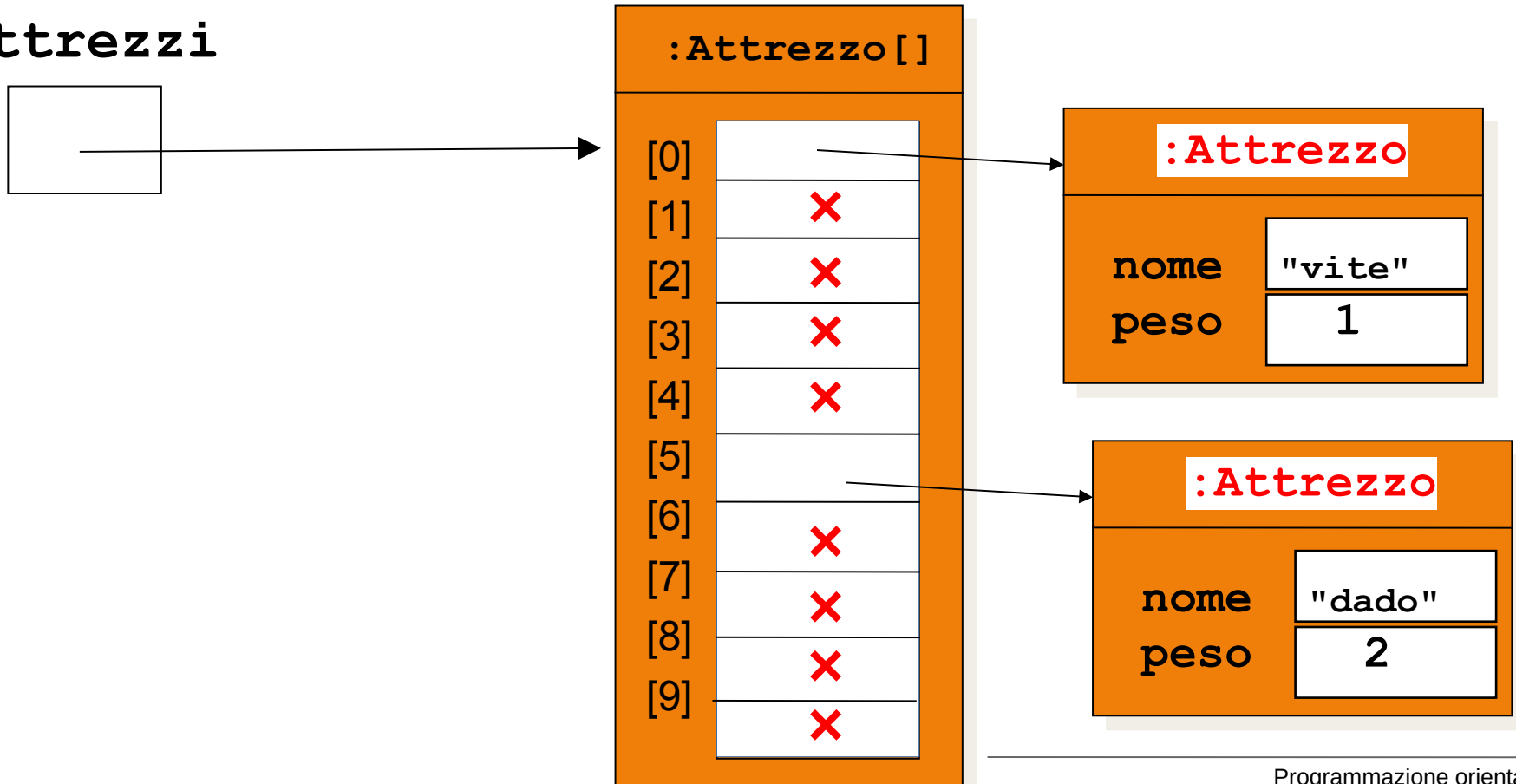
attrezzi



# Array: Diagramma degli Oggetti

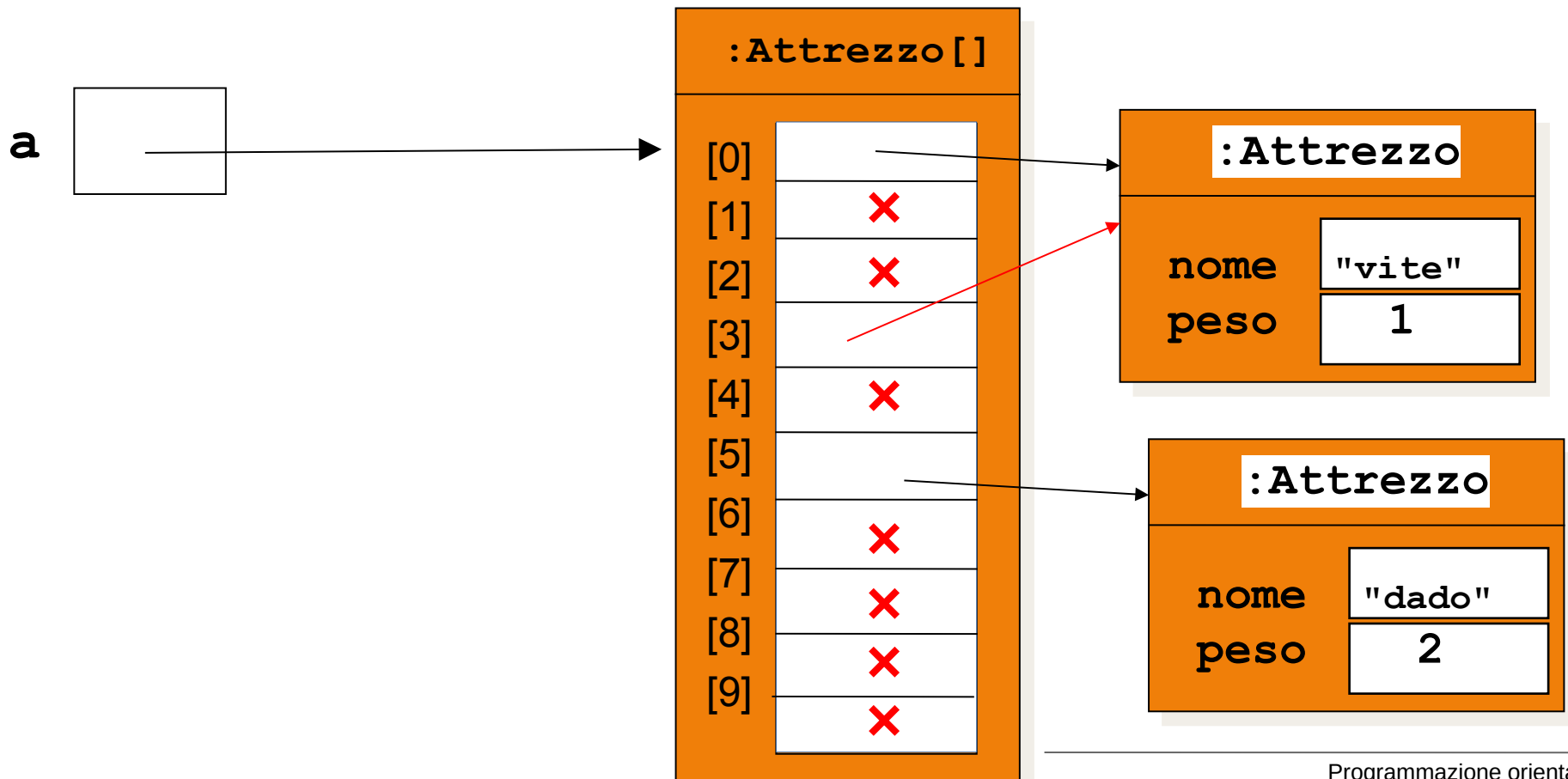
```
Attrezzo attrezzi[];  
attrezzi = new Attrezzo[10];  
attrezzi[0] = new Attrezzo("vite",1);  
attrezzi[5] = new Attrezzo("dado",2);
```

attrezzi



# Array: Diagramma degli Oggetti

```
Attrezzo[] attrezzi;  
attrezzi = new Attrezzo[10];  
attrezzi[0] = new Attrezzo("vite",1);  
attrezzi[5] = new Attrezzo("dado",2);  
attrezzi[3] = attrezzi[0];
```



# Esercizio

- Disegnare il diagramma degli oggetti che rappresenta lo stato del seguente programma al termine della esecuzione della istruzione in linea 7
- Quale valore ha il peso dell'attrezzo referenziato dalla variabile con indice 0 dell'array al termine della istruzione 6? E al termine dell'istruzione 7?

```
1.  Attrezzo[] attrezzi;  
2.  attrezzi = new Attrezzo[5];  
3.  Attrezzo a = new Attrezzo("spada", 40);  
4.  Attrezzo b = new Attrezzo("scudo", 30);  
5.  attrezzi[0] = a;  
6.  attrezzi[1] = b;  
7.  a = b;
```

# Esercizio

- Disegnare il diagramma degli oggetti che rappresenta lo stato del seguente programma al termine della esecuzione della istruzione in linea 7
- Quale valore ha il peso dell'attrezzo referenziato dalla variabile con indice 0 dell'array al termine della istruzione 6? E al termine della istruzione 7?

```
1. Attrezzo[] attrezzi;  
2. attrezzi = new Attrezzo[5];  
3. Attrezzo a = new Attrezzo("spada", 40);  
4. Attrezzo b = new Attrezzo("scudo", 30);  
5. attrezzi[0] = a;  
6. attrezzi[1] = b;  
7. attrezzi[0] = attrezzi[1];
```

# Esercizio (*con Eclipse*)

- Riscrivere la classe **Stanza** in modo che contenga un array di **Attrezzi**
  - Implementare quindi il metodo:  
`boolean addAttrezzo(Attrezzo attrezzo)`  
aggiunge un attrezzo nella stanza.
  - Se c'è spazio restituisce **true**; altrimenti restituisce **false**
  - quindi aggiungere il metodo  
`boolean hasAttrezzo(String nomeAttrezzo)`  
Controlla che nella stanza ci sia un attrezzo di nome `nomeAttrezzo`:
    - Se presente restituisce **true**; altrimenti **false**

# Esercizio

```
public class Stanza {
    private Attrezzo[] attrezzi;
    private int numeroAttrezzi;
    private String nome;
    public Stanza(String nome) {
        this.nome = nome;
        this.attrezzi = new Attrezzo[10]; // per ora solo 10 attrezzi
        this.numeroAttrezzi = 0;
    }
    public boolean addAttrezzo(Attrezzo attrezzo) {
        if (this.numeroAttrezzi < 10) { // massimo 10 attrezzi
            this.attrezzi[numeroAttrezzi] = attrezzo;
            this.numeroAttrezzi++;
            return true;
        } else {
            return false;
        }
    }
    // ... continua ...
}
```



# Esercizio

```
...  
  
public boolean hasAttrezzo(String nomeAttrezzo) {  
    boolean trovato;  
    trovato = false;  
    for (Attrezzo attrezzo : this.attrezzi) {  
        if (attrezzo.getNome().equals(nomeAttrezzo))  
            trovato = true;  
    }  
    return trovato;  
}  
  
} // fine classe Stanza
```

- L'equivalenza di stringhe viene controllata con il metodo `equals()`

# Esercizio

- Scrivere il metodo

`Attrezzo getAttrezzo(String nomeAttrezzo)`

che restituisce l'attrezzo di nome `nomeAttrezzo` se esiste, `null` altrimenti

- Scrivere il metodo

`String toString()`

che restituisca una descrizione della stanza compresa una lista di tutti gli oggetti in essa contenuti

- Conviene scrivere un metodo `toString()` anche nella classe `Attrezzo`

# Costanti

- Talvolta, vogliamo imporre che i valori di alcune variabili di istanza non possano essere cambiati
  - definiamo valori costanti
- A tal fine si usa la parola chiave `final`

```
final double NUMERO_MASSIMO_DIREZIONI = 4;  
NUMERO_MASSIMO_DIREZIONI = 20; // ERR. COMPILAZIONE
```

- Convenzione di stile: gli identificatori delle costanti si scrivono in maiuscolo

# Variabili di Classe

- Talvolta è utile avere variabili che devono essere condivise da tutti gli oggetti della stessa classe: variabili di classe
- A tal fine si usa la parola chiave **static**

```
private static int perTutti;
```

- **ATTENZIONE:** Esistono importanti motivazioni didattiche per limitarne il più possibile l'uso
- Per il momento limitiamoci ad usare questa parola chiave solo ed esclusivamente per
  - dichiarare il metodo **main()**
  - definire costanti

# Costanti e Variabili di Classe

- È ragionevole che una costante sia anche un variabile di classe. Perché?
- Per questo le dichiarazioni delle costanti di solito sono come segue:

```
final static double NUMERO_MASSIMO_DIREZIONI = 4;
```

# Metodi di Classe (Statici)

- Attenzione: nella POO i metodi di classe sono una vera e propria *anomalia*
  - Durante l'apprendimento del paradigma OO sono quasi “pericolose”!
- Un metodo di classe corrisponde ad una operazione che può essere svolta senza utilizzare lo stato dell'oggetto (oppure usando solo variabili di classe, condivise da tutti gli oggetti)
- Tipicamente si usano per realizzare funzioni pure (es. i metodi di `java.lang.Math`)
- Sono come il freno a mano di un'automobile:
  - Molto utile da fermi...
  - ...ma non usatelo mai per frenare in corsa!!

# Ancora su final (1)

- Attenzione:

```
final int a = 10;  
a = 3; // ERRORE DI COMPILAZIONE
```

```
final int a = 10;  
int b = 4;  
a = b; // ERRORE DI COMPILAZIONE
```

# Ancora su final (2)

- Attenzione:

```
final Stanza ds1 = new Stanza("Aula DS1");  
Stanza n7 = new Stanza("Aula N7");  
  
ds1 = n7; // ERRORE DI COMPILAZIONE
```

```
final Stanza ds1 = new Stanza("Aula DS1");  
Stanza n7 = new Stanza("Aula N7");  
Attrezzo v = new Attrezzo("vite",1);  
  
ds1.setAttrezzo(v); // LECITO! PERCHE' ?
```



# Ancora su `final` (3)

- In sostanza, quando `final` si usa su
  - primitivi: rende costante la variabile
  - riferimenti: rende costante il riferimento, ma non il contenuto dell'oggetto referenziato, che rimane libero di cambiare

# Studio di Caso (1)

- Il gioco di ruolo **diadia**
  - In questa versione iniziale ci concentriamo su poche classi
  - La prima versione, oltre ad essere molto semplice, ha un codice scritto piuttosto male:
    - Ci sono errori (a tempo di esecuzione)
    - È di difficile manutenzione
    - È poco riutilizzabile
  - Il primo aspetto (errori) lo verificherete immediatamente eseguendo il programma
  - Capire come ovviare agli altri due aspetti è uno degli obiettivi formativi del corso

# Studio di Caso (2)

- Scaricare lo studio di caso versione base:
  - <https://sites.google.com/view/rm3-poo/materiale-didattico>
- Eseguire il metodo **main()** nella classe **DiaDia**
- Digitare alcuni comandi
  - **aiuto** per avere un elenco dei comandi
  - **vai nord|sud|est|ovest** per cambiare stanza
- Spostandoci da una stanza all'altra noteremo presto errori (uno porta alla terminazione del programma)

# Esercizio

- Studiare a fondo il codice della classe `Stanza`
  - a che cosa serve la variabile `numeroDirezioni`?
  - che cosa fanno i metodi `impostaStanzaAdiacente(String, Stanza)` e `getStanzaAdiacente(String)`?

# Esercizio

- Scrivere una classe `StanzaTest1` con il metodo `main()` con le istruzioni per:
  - Definire due oggetti `Stanza`: *bar* e *mensa*
  - Impostare le uscite dei due oggetti affinché:
    - L'uscita nord del *bar* porti nella *mensa*
    - L'uscita sud della *mensa* porti nel *bar*
  - Stampare la descrizione della stanza dietro la porta nord del *bar*
  - Stampare la descrizione della stanza dietro la porta sud della *mensa*
- ✓ Controllare che le stampe siano quelle attese

# Esercizio

- Scrivere una classe **StanzaTest2** con un metodo **main()**:
  - Definire due oggetti **Stanza**: *bar* e *mensa*
  - Definire due oggetti **Attrezzo**: *tazzina* e *piatto*
  - Impostare le uscite dei due oggetti **Stanza** affinché:
    - L'uscita nord del *bar* porti nella *mensa*
    - L'uscita sud della *mensa* porti nel *bar*
  - Aggiungere nel *bar* l'oggetto **Attrezzo** *tazzina*
  - Aggiungere nella *mensa* l'oggetto **Attrezzo** *piatto*
  - Stampare il nome e il peso dell'attrezzo presente nella stanza dietro la porta nord del *bar*
  - Stampare il nome e il peso dell'attrezzo presente nella stanza dietro la porta sud della *mensa*
  - ✓ Controllare che le stampe siano quelle attese