

Valori di default delle variabili istanza

boolean false

byte, short 0
int, long

float, double 0

char '0000' tutto minuscolo

riferimenti null

class A {

```
int x, y;  
double a = 3.14;  
double b = a + 1.0;
```

double c = b + 1.1 + Math.sqrt(3.5);

:

↓ ordine di
inizializ.

A metà non devono
lanciare eccezioni di
tipo "checked"

}

Costruttori

Sequenze di istruzioni utilizzate per definire
lo stato di un oggetto

- quando i meccanismi di initializzazione
non sono sufficienti

- quando i valori da usare in fase di inizializzazione sono molti solo nel momento in cui si fa la "new"

In Java un costruttore

- ha lo stesso nome della classe
- non restituisce alcun valore
- può essere eseguito solo insieme a "new"
- può essere obbligato da zero o più argomenti

Se il programmatore definisce una classe privo di costruttori allora è automaticamente presente un costruttore privo di argomenti e che non fa niente

(il costruttore di default)

Esempio:

```
class Pippo {
    int a;
}
// metodi, ma non costruttori
Pippo p1 = new Pippo();
```

 **Costruttore di default**
OK

Se ci sono più costruttori allora devono essere distinguibili attraverso il numero e/o il tipo degli argomenti

```
class A {  
    double a, b;
```

```
A (double c, double d) {  
    a = c;  
    b = d;  
}
```

```
A (double e) {  
    a = e;  
}  
// metodi  
}
```

A a1 = new A(1.1, 2.2);

A a2 = new A(5.0);

1° costruttore
2° Costruttore

Se il programmatore definisce almeno un costruttore allora il costruttore di default non c'è più (ma il programmatore può in modo esplicito definire uno privo di argomenti)

A a3 = new A(); // errore il costruttore
// di default non c'è

```
class C {  
    int x;  
    C (int miox) {  
        x = miox;
```

```

C( int miox) {
    x = miox;
}
C() {
    x = 50;
}
:
{

```

C c1 = new C(); // OK

C c2 = new C(20); // OK

Metodi

Java supporta overloading dei metodi:

ci possono essere più metodi con lo stesso nome

olovemo però essere distinguibili grazie al tipo e/o al numero dei parametri

il tipo del valore restituito non è utile a distinguere un metodo dall'altro

Esempio:

```

class Punto {
    double x,y;
}

Punto ( double a , double b ) {
    x = a;
    y = b;
}

void trasla ( double z ) {
    x = x + z;
    y = y + z;
}

```

arguments
formal

1 - 7^c)

}

void trasla (double x, double y) {

x = x + 2;

y = y + 4;

}

double distanza () {

return Math.sqrt (x * x + y * y);

}

}

argomento attuale

Punto p1 = new Punto (1.0, 2.0);

p1.trasla (5.1); // viene invocato il primo
// metodo trasla, quello con
// un solo argomento

p1.trasla (3.4, 5.7); // viene invocato il
// secondo metodo trasla
// quello con due argomenti

Quando invochiamo un metodo
dobbiamo fornire un valore (l'argomento
attuale) per ognuno degli argomenti formal.

Passaggio dei parametri

In Java il passaggio è sempre
per valore

class Esempio {

void m (int x) {

x ++;

}

3
3

Esempio `es = new Esempio();`
`int y = 10;`
`es.m(y);`
quanto vale y ?
 y vale 10

Ogni volta che un metodo viene chiamato sullo stack viene creato un record di attivazione

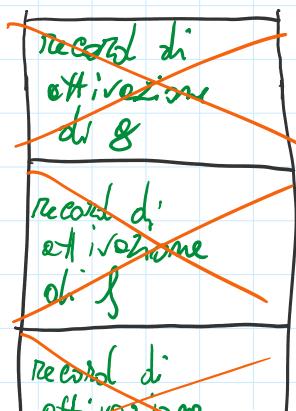
record di attivazione: una struttura dati allocata sullo stack in cui

- viene riservato lo "spazio" per le variabili locali del metodo
- viene riservato lo "spazio" per gli argomenti formali
- gli argomenti formali vengono inizializzati con i valori dei corrispondenti argomenti attuali
- al termine dell'esecuzione di un metodo il record di attivazione viene distrutto

... `main(...)` {
 `f();`

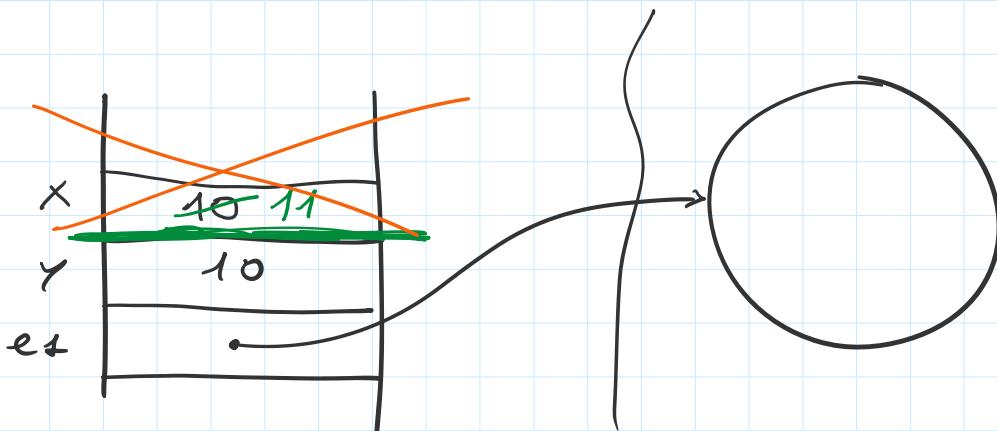
}

.... `f()` {



```
.... f() {  
    :  
    g();  
:  
}
```

~~record di
effettuazione
del main~~



Vediamo cosa succede nel caso di argomenti
di tipo riferimento

```
class Uno {  
    int valore;
```

```
Uno (int v){  
    valore = v;  
}
```

```
void setValore (int v){  
    valore = v;  
}
```

}

```
class Due {
```

```
void m1 (Uno u){  
    u = new Uno(1000);  
}
```

```
void m2 (Uno u){
```

u. setVidre(50);

}

Uno u1 = new Uno(10);

Due d1 = new Due();

d2. m1(u1);

d1. m2(u1);

=

- il metodo chiamato non può modificare il valore di una variabile del tipo riferimento del chiamante
- il metodo chiamato può modificare il valore di un oggetto di cui ha ricevuto il riferimento dal chiamante

Variabili locali

Non è necessario definirle all'inizio del metodo

Vengono dichiarate quando termina l'esecuzione del metodo

Non hanno un valore di default

Il compilatore controlla che ogni variabile locale abbia un valore ben definito prima del suo utilizzo

```
void m() {
    int x;
    x++; // errore
}
```

```
void m(boolean b) {
    int x;
    if (b) x = 10;
    else   x = 20;
    x++; // OK
}
```

3

Capita che il compilatore non ha abbastanza intelligenza da capire che, qualunque sia l'eccezione, un valore viene sempre dato a una variabile locale.

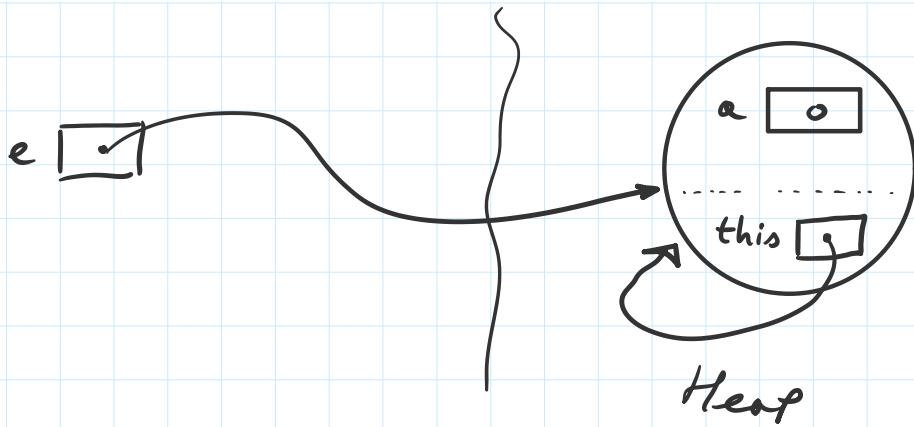
- in questo caso è necessario fornire un valore alla variabile locale

this

La parola chiave `this` può essere usata come riferimento all'oggetto corrente (l'oggetto implicito)

```
class Esempio {  
    int a;  
    void incrementa(){  
        a++;  
    }  
    void m(){  
        incrementa();  
    }  
}
```

Esempio e = new Esempio();



Riscriviamo il codice di Esempio
facendo uso di this

```
class Esempio {
    int a;
    void incrementa() {
        this.a++;
    }
    void m() {
        this.incrementa();
    }
}
```

Attenzione:
non scrivere
codice in questo
modo
è uno stile
inutilmente
proliso e quindi
meno leggibile

Quando dobbiamo usare this?

- quando è necessario passare o restituire un riferimento all'oggetto implicito
- quando ci sono delle omonimità e this ci permette di risolvere le ambiguità

```
class Punto {
    double x;
    double y;
```

```
Punto(double a, double b){}
```

Punto (double x , double y) {

 x = a;
 y = b;
}

:

}

è meglio scrivere

class Punto {

 double x;
 double y;

Punto (double x , double y) {

questa
x è

this.x = x;
this.y = y;

questa x è

}

:

}

Supponiamo di avere una classe Numero

class Numero {

 int n;

 Numero (int n) {

 this.n = n;

}

 Numero maggiore (Numero altro) {

 if (n > altro.n)

 return this;

 else

 return altro;

}

}

```

void stampa() {
    System.out.println(n);
}

```

```

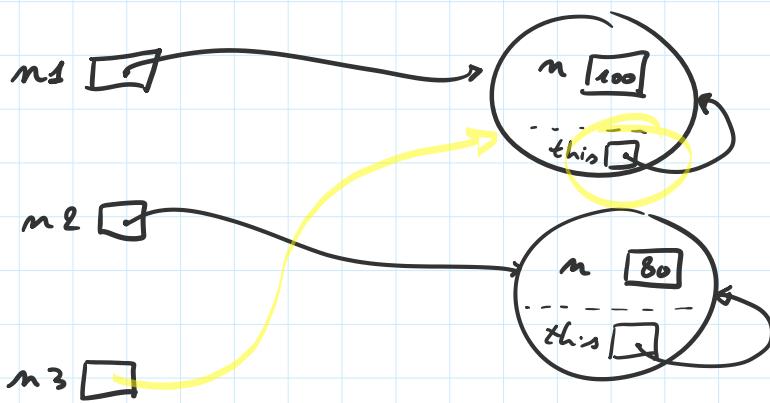
Numero n1 = new Numero(100);
Numero n2 = new Numero(80);
Numero n3;

```

```

n3 = n1.maggiore(n2);
n3.stampa(); // stampa 100

```



this può essere utile anche per evitare di avere codice duplicato nei costruttori!

```
class Studente {
```

```

    String nome;
    String cognome;
    int matricola;
    double media;

```

```
Studente(String nome, String cognome,
         int matricola, double media) {
```

```

    this.nome = nome;
}
```

this.nome = nome;
this.cognome = cognome;
this.matrcola = matrcola;
this.media = media;

}

Studente (String nome, String cognome
int matrcola) {

this.nome = nome; } Coda
this.cognome = cognome; } implicita
this.matrcola = matrcola

}

:

}

Quando definiamo un costruttore
possiamo richiamarne un altro

this(....)

è un parametro per il costruttore
da chiamare.



che è dove è
la prima istruzione
del costruttore che stiamo definendo

il secondo costruttore lo possiamo
scrivere così:

Studente (String nome, String cognome
int matrcola) {

 this(nome, cognome, matrcola, 0.0);

}

dopo this(...) posso
avere altre istruzioni

Campi e metodi statici

Campi statici: utili a memorizzare informazioni relative a tutta la classe e non ai singoli oggetti

Un campo statico esiste in copie singole, indipendentemente dal numero di oggetti della classe

class Veicolo {

```
String proprietario;  
int id;  
static int contatore = 1;
```

Veicolo (String p) {

```
proprietario = p;  
id = contatore++;
```

}

void stampa () {

```
System.out.println(proprietario + " " + id);
```

}

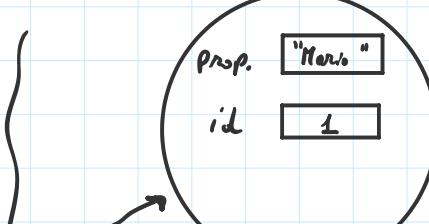
public static void main (String [] args) {

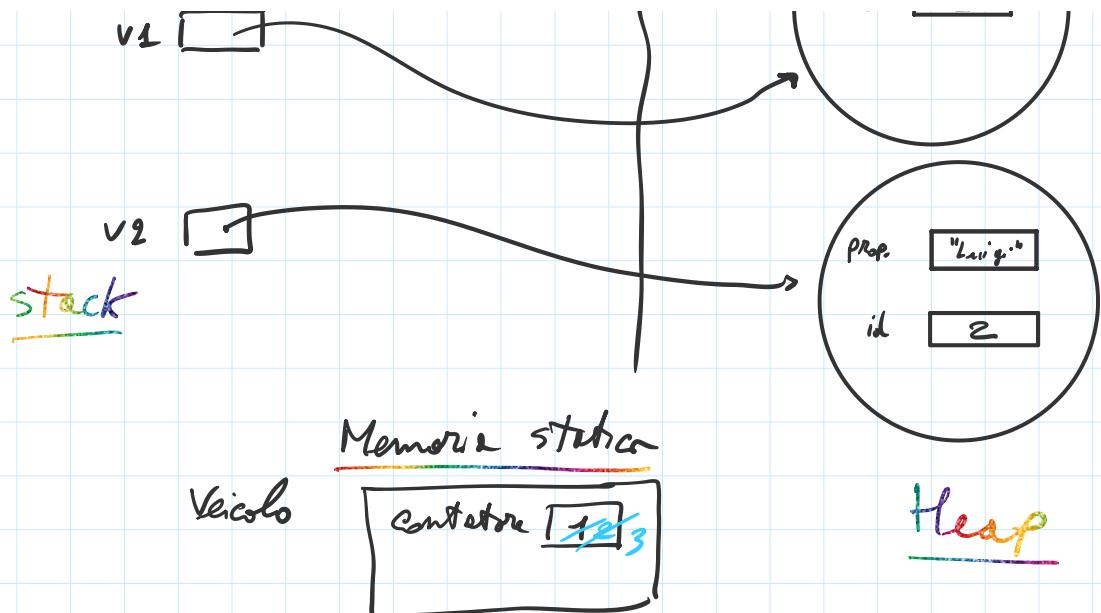
```
Veicolo v1 = new Veicolo ("Mario");  
Veicolo v2 = new Veicolo ("Luigi");  
v1.stampa();  
v2.stampa();
```

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

}

v1





3

In generale per accedere a un campo statico:

NomeDellaClasse. nomeDelCampoStatico

Per esempio:

```
class Principale {
```

```
    public static void main (String [] args) {
```

```
        int q = Veicolo. Contatore;
```

```
    }
```

```
}
```

Veicolo v3 = new Veicolo("Bravo");
q = v3. Contatore; // poco chiaro che è un campo statico (avitor)

I campi statici hanno gli stessi valori di default dei campi non statici

Anche per i campi statici è possibile fornire inizializzatori

class Veicolo {

 static int contatore = 1;

L'inizializzatore di un campo statico può usare

- letterali
- altri campi di tipo static
- valori restituiti da metodi static

Non può usare

- variabili istanza (campi non static)
- metodi non static

class A {

```
int x;  
static int y = 1; // OK: letterale  
static int z = y + 3; // OK  
static int w = x + 2; // ERRORE
```

↑

x di chi?

di quale oggetto?

```
static int k = y + 1 + metodoStatic();
```