

# 13 - Metodi e Costruttori

Programmazione e analisi di dati  
Modulo A: Programmazione in Java

Paolo Milazzo

Dipartimento di Informatica, Università di Pisa  
<http://pages.di.unipi.it/milazzo>  
milazzo@di.unipi.it

Corso di Laurea Magistrale in Informatica Umanistica  
A.A. 2018/2019

# Dichiarazione di una classe

Lo schema generale di dichiarazione di una classe in Java è il seguente:

```
public class <nome-classe> {  
    <variabili di istanza>  
    <variabili statiche>  
    <costruttori>  
    <metodi di istanza>  
    <metodi statici>  
}
```

- Variabili e metodi (siano essi d'istanza o statici) sono chiamati anche **membri** della classe
- L'ordine delle dichiarazioni all'interno del corpo di una classe non è importante

# Membri statici o di istanza? (1)

Abbiamo visto che si usano

- **membri d'istanza** per codificare lo stato e le funzionalità dei **singoli oggetti**
- **membri statici** per codificare lo stato e le funzionalità **della classe**
  - ▶ condivise da tutti gli oggetti
  - ▶ significativi anche se non esiste nessun oggetto della classe

## Membri statici o di istanza? (2)

Quindi:

- Il metodo `main` è statico. In effetti viene invocato prima di creare qualunque oggetto.
- Una variabile dovrebbe essere d'istanza se assume valori diversi per oggetti diversi.
  - ▶ Esempio: base e altezza di un oggetto Rettangolo
- Una variabile dovrebbe essere statica se assume gli stessi valori per oggetti diversi.
  - ▶ In caso contrario potrebbe essere difficile mantenere lo stesso valore per tale variabile in tutti gli oggetti.
- Se un metodo utilizza una variabile d'istanza non può essere statico
  - ▶ Il compilatore segnalerebbe un errore
  - ▶ Il modificatore `static` per i metodi serve solo per far controllare al compilatore di non usare variabili d'istanza
- Se un metodo non utilizza variabili d'istanza dovrebbe essere statico
  - ▶ In caso contrario il compilatore non segnala errori, ma il programma potrebbe essere concettualmente poco chiaro

## Modificatori di visibilità

Abbiamo visto che una dichiarazione di variabile o metodo (sia statico che di istanza) può essere preceduto da **modificatori di visibilità**.

In ordine crescente di visibilità abbiamo:

<code>private</code>	Utilizzabile solo all'interno della stessa classe
senza modificatore	Utilizzabile solo nel <b>package</b> che contiene la classe
<code>protected</code>	Utilizzabile nel <b>package</b> che contiene la classe, e in tutte le classi che <b>ereditano</b> da essa
<code>public</code>	Utilizzabile ovunque

Vedremo più avanti i **package** e il concetto di **ereditarietà**

# Interfaccia pubblica di una classe

I membri pubblici di una classe (compresi i costruttori) costituiscono l'**interfaccia pubblica** della classe

- E' l'insieme delle risorse e delle funzionalità messe a disposizione delle altre classi

Invece:

- Le **variabili private** di una classe rappresentano lo **stato interno** della classe
- I **metodi privati** di una classe sono **metodi ausiliari** a disposizione degli altri metodi della classe

# Metodi (1)

La dichiarazione di un metodo ha la seguente sintassi:

```
<modificatori> <tipo> <nome> (<lista_parametri_formali>) {  
    ....  
}
```

Esempio:

```
public static int minimo(int a, int b) {  
    if (a<b) return a;  
    else return b;  
}
```

- **Modificatori:** di visibilità (*private*, *public*, ...) o di appartenenza a classe o istanze (*static*)
- **Tipo:** del risultato restituito dal metodo
- **Nome:** del metodo
- **Parametri formali:** lista (anche vuota) di coppie  
<tipo> <parametro>

## Metodi (2)

I metodi e la possibilità di definire variabili private consentono di limitare l'accesso allo stato di un oggetto

```
public class SolaLettura {  
    private int valore;  
  
    public SolaLettura(int x) {  
        valore=x;  
    }  
  
    public int getValore() {  
        return valore;  
    }  
}
```

La variabile d'istanza `valore` potrà essere acceduta in sola lettura



## Metodi (3)

Tramite i metodi si può anche controllare la scrittura delle variabili private

```
public class SolaLettura {  
    private int valore;  
  
    public SolaLettura(int x) {  
        valore=x;  
    }  
  
    public int getValore() {  
        return valore;  
    }  
  
    // modifica il valore solo se il nuovo e' maggiore  
    // restituisce true se la modifica e' stata effettuata  
    public boolean setValore(int nuovo) {  
        if (nuovo>valore) {  
            valore=nuovo;  
            return true;  
        }  
        else  
            return false;  
    }  
}
```

La modifica della variabile `valore` sarà sotto controllo del metodo!

# Incapsulamento (1)

La possibilità di controllare l'accesso allo stato degli oggetti tramite appositi metodi nei linguaggi orientati agli oggetti prende il nome di

## Proprietà di Incapsulamento

L'incapsulamento consente di **nascondere** la rappresentazione dello stato interno degli oggetti agli utilizzatori

- L'utilizzatore di `getValore` e `setValore` non è necessariamente a conoscenza che il valore è rappresentato nella classe da una variabile di tipo `int`
- La classe potrebbe ad esempio sfruttare un array di `int` che memorizza (all'insaputa dell'utilizzatore) gli ultimi 10 valori ricevuti...

## Incapsulamento (2)

Abbiamo già discusso che l'incapsulamento è una proprietà essenziale per consentire lo sviluppo di **applicazioni complesse**

- E' sufficiente i vari programmatori si accordino sulle interfacce pubbliche delle varie classi che ognuno deve implementare

L'utilizzo della terminologia `get...` e `set...` per i metodi che consentono di accedere a variabili private è una prassi comune:

- `getValue()`, `setValue(x)`, `getFirstElement()`  
`getLastElement()`, ...

# Passaggio dei parametri

In Java il passaggio dei parametri ai metodi avviene **per valore**

- I metodi lavorano su **copie** delle variabili passate come parametri

E quando si **passano oggetti** ai metodi?

Le variabili di tipo classe contengono **riferimenti** a oggetti

- Ciò che viene copiato al momento della chiamata è il riferimento!
- Il metodo lavora sull'**oggetto originale** (acceduto tramite la copia del riferimento)

# Riferimenti e incapsulamento: ATTENZIONE (1)

Assumendo che la classe Rettangolo sia la seguente:

```
public class Rettangolo {  
    public int base;  
    public int altezza  
    public Rettangolo(int b, int a) { base = b; altezza = a; }  
}
```

Ragioniamo sulla seguente classe che usa Rettangolo:

```
public Finestra {  
  
    // variabile a cui non vogliamo dare accesso pubblico  
    private Rettangolo area;  
  
    public Finestra() {  
        area = new Rettangolo(800,600);  
    }  
  
    public Rettangolo getArea() {  
        return area;  
    }  
}
```

La variabile privata area è al sicuro?

## Riferimenti e incapsulamento: ATTENZIONE (2)

Risposta: NO

Vediamo cosa potrebbe fare un utilizzatore della classe Finestra

```
public class UsaFinestra {
    public static void main(String[] args) {
        Finestra f = new Finestra();
        Rettangolo r = f.getArea();
        System.out.println(r.base); // stampa 800
        r.base=100;

        Rettangolo r2 = f.getArea();
        System.out.println(r2.base); // stampa 100
    }
}
```

Il metodo `getArea()` di `Finestra` restituisce un riferimento all'oggetto privato!!!

# Overloading (1)

Java supporta un meccanismo di **overloading** (**sovraccarico**) che consente di chiamare più metodi della stessa classe con lo stesso nome, purché ogni metodo abbia una diversa **firma** (**signature**).

Dato un generico metodo:

```
<modificatori> <tipo> <nome> ( <tipo1> <p1> , ... , <tipon> <pn> ) {  
    ...  
}
```

la firma corrispondente è data dalla sequenza:

```
<nome>(<tipo1>, ..., <tipon>)
```

**Attenzione:** la firma non comprende né il **tipo** del metodo, né i **nomi** dei parametri formali.

## Overloading (2)

Esempi:

<b>Metodo</b>	<b>Firma</b>
<code>int getVal()</code>	<code>getVal()</code>
<code>int minimo(int x, int y)</code>	<code>minimo(int,int)</code>
<code>int minimo(int a, int b)</code>	<code>minimo(int,int)</code>
<code>double minimo(double x, double y)</code>	<code>minimo(double,double)</code>
<code>int minimo(int x, int y, int z)</code>	<code>minimo(int,int,int)</code>

Quali di questi metodi potrebbero stare nella stessa classe?



## Overloading (3)

Esempio:

```
public class Sommatore {  
  
    public static int somma(int x) {  
        return x;  
    }  
  
    public static int somma(int x, int y) {  
        return x+y;  
    }  
  
    public static int somma(int x, int y, int z) {  
        return x+y+z;  
    }  
  
}
```

La possibilità di usare l'**overloading** di un metodo consente di usare lo stesso nome per metodi diversi che realizzano **la stessa funzionalità** su dati di tipo diverso.

# Varargs (1)

Recentemente (più o meno) in Java è stata introdotta la possibilità di definire metodi con un **numero variabile di parametri**

- Metodi con **varargs**

Nella **definizione** di un metodo con varargs l'**ultimo** parametro formale ha la seguente sintassi:

```
<tipo>... par
```

dove i puntini sospensivi danno l'idea che il **parametro attuale corrispondente** può essere presente **zero o più volte**.

All'interno del corpo del metodo, il parametro par avrà tipo `<tipo> []`....

Attenzione: solo l'ultimo parametro formale di un metodo può avere questa nuova sintassi.

## Varargs (2)

Esempio:

```
public class Sommatore {  
  
    public static int somma(int... valori) {  
        int s = 0;  
        for (int x: valori) {  
            s += x;  
        }  
        return s;  
    }  
}
```

Utilizzo:

```
System.out.println(somma()); //stampa 0  
System.out.println(somma(3,4)); //stampa 7  
System.out.println(somma(5,7,2,1,6,8,1)); //stampa 30
```

La firma di `int somma(int... valori)` è `somma(int[])`

# Costruttori e inizializzazione di variabili

Fino ad ora abbiamo visto che un **costruttore** è un metodo speciale che viene eseguito al momento della creazione di un oggetto (tramite la primitiva **new**

- Tipicamente **inizializza le variabili di istanza** dell'oggetto

L'inizializzazione delle variabili in Java merita un discorso...

# Inizializzazione di variabili (1)

Abbiamo visto due tipologie di variabili in Java

- variabili dichiarate localmente nei metodi (o costruttori): **variabili locali** e **parametri formali**
- variabili dichiarate come membri di una classe: **variabili statiche** e **variabili d'istanza**

## Inizializzazione di variabili (2)

Le **variabili locali** dei metodi:

- vengono allocate nel record di attivazione relativo alla chiamata del metodo (nello **stack**)
- devono essere **inizializzate esplicitamente**, altrimenti il compilatore segnala un errore

Esempio:

```
int num;
while (num >= 0) {
    num = input.nextInt();
}
```

Errore segnalato dal compilatore:

```
variable num might not have been initialized
```

## Inizializzazione di variabili (3)

Le **variabili statiche e di istanza**:

- vengono allocate nell'area di memoria che descrive l'oggetto (nello **heap**)
- vengono **sempre inizializzate**, anche se non viene fatto esplicitamente

I valori di default di queste variabili sono:

- 0 per le variabili numeriche
- false per le variabili di tipo boolean
- null per le variabili di tipo classe (oggetti)

Ci sono **tre modi** per inizializzare una variabile d'istanza o statica:

- Con assegnamento esplicito all'interno di un costruttore;
- Con inizializzazione esplicita nella dichiarazione;
- Inizializzazione (implicita) con valori default.

## Inizializzazione di variabili (4)

Esempio in cui tutti e tre i modi sono impiegati:

```
public class Punto3d {  
  
    public double x;           // inizializzazione nel costruttore  
    public double y = 3.0;    // inizializzazione nella dichiarazione  
    public double z;         // inizializzazione con valore di default  
  
    public Punto3d(double val) {  
        x = val;  
    }  
}
```

Esempio di uso

```
Punto3d p = new Punto3d(5.0);  
  
System.out.println(p.x); // stampa 5.0;  
System.out.println(p.y); // stampa 3.0;  
System.out.println(p.z); // stampa 0.0;
```



## Costruttori (1)

I costruttori si dichiarano all'interno di una classe essenzialmente come i metodi, ma rispettando le seguenti **regole**:

1. il nome del costruttore deve coincidere con quello della classe;
2. il tipo del costruttore non deve essere specificato.
3. il modificatore `static` non può essere usato per un costruttore.

Come per i metodi, anche ai costruttori si applica l'**overloading**:

- una classe può avere più costruttori purché abbiano firma diversa (cioè i parametri formali devono differire nel tipo e/o nel numero).

Esempio (classe `ContoCorrente`):

```
public ContoCorrente(double saldoIniziale) {
    saldo = saldoIniziale;
}

public ContoCorrente() {
    saldo = 0.0;
}
```

## Costruttori (2)

Ogni classe ha un **costruttore di default** che inizializza le variabili d'istanza con il corrispondente valore di default

Questo costruttore non ha parametri, ed è disponibile **solo se nella classe non è definito nessun costruttore**

Se invece nella classe è definito **almeno un costruttore**, allora il costruttore di default **non è più utilizzabile**

Esempio di classe senza costruttori:

```
public class Punto {  
    public double x, y;  
    ... // altri metodi ma  
    ... // nessun costruttore  
}
```

Viene utilizzato il costruttore di default:

```
Punto p = new Punto(); // OK  
System.out.println(p.x); // stampa 0.0;  
System.out.println(p.y); // stampa 0.0;
```

## Costruttori (3)

Esempio di classe con costruttori:

```
public class Punto {  
    public double x, y;  
  
    public Punto(double vx, double vy) {  
        x = vx;  
        y = vy;  
    }  
    ... // altri metodi  
}
```

Il costruttore di default non si può usare:

```
Punto p1 = new Punto(); // ERRORE  
Punto p2 = new Punto(3.0, 5.0) // OK
```

## Costruttori (4)

Se il programmatore vuole continuare a rendere disponibile un costruttore che non prevede parametri, deve implementarlo:

```
public class Punto {  
    public double x, y;  
    public Punto() {} // inizializzazione automatica  
    public Punto(double vx, double vy) {  
        x = vx;  
        y = vy;  
    }  
    ... // altri metodi  
}
```

Questa volta si possono usare entrambi i costruttori

```
Punto p1 = new Punto(); // OK  
Punto p2 = new Punto(3.0, 5.0) // OK
```

# Il riferimento “this” (1)

In ogni corpo di un metodo d'istanza (o di un costruttore) è sempre disponibile la variabile `this`

- è un riferimento all'oggetto su cui si invoca il metodo (o costruttore)
- è detto anche `parametro implicito` del metodo

## Il riferimento "this" (2)

Il riferimento `this` può essere usato per accedere alle variabili di istanza del metodo:

Senza `this`

```
public class Punto {  
    public double x, y;  
  
    public String toString() {  
        return "(" + x + "," + y + ")";  
    }  
}
```

Con `this` (equivalente al precedente)

```
public class Punto {  
    public double x, y;  
  
    public String toString() {  
        return "(" + this.x + "," + this.y + ")";  
    }  
}
```

## Il riferimento “this” (3)

In realtà `this` in alcuni casi è utile:

Consente di usare come nome di parametro formale lo stesso nome di una variabile d'istanza

```
public class Punto {  
    public double x, y;  
  
    public Punto(double x, double y) {  
        // assegna i parametri alle variabili d'istanza  
        this.x = x;  
        this.y = y;  
    }  
}
```

L'uso di `this` può rendere più chiara la lettura del codice di classi complesse

- e' subito chiaro che si sta usando una variabile d'istanza

## Il riferimento "this" (4)

this consente anche di scrivere metodi che restituiscono un riferimento all'oggetto corrente:

```
public class Punto {  
    public double x, y;  
    ... // costruttori  
    // restituisce il "minimo" tra p e l'oggetto corrente  
    public Punto minimo(Punto p) {  
        if ( (p.x+p.y) < (this.x+this.y))  
            return p;  
        else  
            return this;  
    }  
}
```

E si usa così'

```
Punto p1 = new Punto(3.0, 7.0);  
Punto p2 = new Punto(10.0, 15.0);  
Punto min = p1.minimo(p2);
```



## Il riferimento “this” (5)

Infine, `this` (o meglio, `this()`) può essere usato da un costruttore per chiamarne un'altro

- **Attenzione:** questo uso è consentito soltanto **come prima istruzione** del costruttore

Senza `this()`

```
public class Punto {  
    public double x, y;  
  
    // prende un solo valore e lo assegna sia a x che a y  
    public Punto(double z) {  
        this.x = z;  
        this.y = z;  
    }  
  
    public Punto(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

## Il riferimento “this” (6)

### Con this()

```
public class Punto {  
    public double x, y;  
  
    // prende un solo valore e lo assegna sia a x che a y  
    public Punto(double z) {  
        this(z,z); // chiama l'altro costruttore  
    }  
  
    public Punto(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```