

Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

Capitolo 6: Classi in Java

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Campi `public` e `private` · Costruttori
Realizzare una classe e i suoi clienti · Il riferimento `this`

Indice

1	Campi <code>public</code> e <code>private</code>	1
2	Costruttori	2
3	Realizzare una classe e i suoi clienti	5
4	Il riferimento <code>this</code>	10
5	Esercizi	12
6	Soluzioni	13

1 Campi **public** e **private**

Definizione – Campi **public** e **private**

I membri di una classe, dati e metodi, si chiamano **campi**. Ogni campo si dichiara con un livello di accesso:

- **public**: il campo è visibile ai **clienti** della classe, cioè al codice delle altre classi;
- **private**: il campo si usa solo nel codice della classe stessa e non è visibile all'esterno.

L'insieme dei campi **public** è l'**interfaccia pubblica** della classe.

```
public class Persona {  
    // campi dati (variabili di istanza)  
    private String nome;  
    private String residenza;  
  
    // campi funzione (metodi)  
    public String getNome() {  
        return nome;  
    }  
    public String getResidenza() {  
        return residenza;  
    }  
    public void setResidenza(String nuovaResidenza) {  
        residenza = nuovaResidenza;  
    }  
}
```

Un cliente di **Persona** può invocare i tre metodi, ma non leggere né scrivere i campi dati: l'accesso a un campo **private** da un'altra classe è un errore di compilazione.

```
Persona p1 = new Persona();  
p1.setResidenza("Roma");           // corretto: setResidenza è public  
System.out.println(p1.getResidenza()); // corretto: stampa Roma  
System.out.println(p1.residenza);    // errore: residenza è private
```

Il compilatore segnala *residenza has private access in Persona*.

Nota – Getter e setter

Il valore di un campo privato si offre ai clienti con metodi pubblici:

- un **getter**, come `getNome()`, restituisce il valore;
- un **setter**, come `setResidenza(String r)`, lo cambia. Di solito restituisce `void` e ha un parametro dello stesso tipo restituito dal getter.

Proprietà mutabili e immutabili. Per ogni proprietà della classe si sceglie se può cambiare durante la vita dell'oggetto:

- una proprietà **mutabile**, come la residenza di una persona, ha getter e setter;
- una proprietà **immutabile**, come il nome, ha solo il getter, e riceve il valore quando l'oggetto nasce, dal costruttore (Sezione 2).

Il getter c'è quasi sempre: una proprietà che nessuno può leggere non serve. Il setter si mette solo se la proprietà deve poter cambiare.

Con i campi privati la classe sceglie liberamente come rappresentare i dati: il nome potrebbe essere memorizzato cifrato, la residenza come latitudine e longitudine. Il cliente vede solo i metodi, e il getter converte la rappresentazione interna nel tipo promesso dall'interfaccia. È l'**information hiding** del Capitolo 2 realizzato dal linguaggio: il codice che tenta di usare la rappresentazione interna non compila.

Nota – Metodi ausiliari

Anche un metodo può essere **private**: è un metodo **ausiliario** (*helper*), che serve a realizzare i servizi pubblici, per esempio spezzando un algoritmo lungo in metodi più piccoli. I clienti vedono solo i servizi.

Principio (Capitolo 2)	Come lo realizza la classe
Information hiding	i campi dati sono private : la rappresentazione non si vede da fuori
Interfacciamento esplicito	su un oggetto si può fare solo ciò che offrono i metodi public , scelti da chi progetta la classe
Alta coesione	dati e metodi di un concetto, come la persona, stanno nella stessa classe

2 Costruttori

Il nome di una **Persona** è privato e non ha setter: un cliente non può assegnarlo, `p.nome = "Luigi Bianchi"` non compila. Lo assegna il costruttore.

Definizione – Costruttore

Un **costruttore** è un metodo speciale che:

- ha lo stesso nome della classe e nessun tipo restituito, neanche **void**;
- gestisce la nascita di un oggetto, inizializzandone i campi dati;
- si invoca solo con **new**, quando l'oggetto viene creato.

```
public Persona(String n, String r) {  
    nome = n;  
    residenza = r;  
}
```

```
Persona p = new Persona("Luigi Bianchi", "Milano");  
System.out.println(p.getNome());           // Luigi Bianchi  
System.out.println(p.getResidenza());       // Milano  
p.setResidenza("Roma");                     // Luigi Bianchi si trasferisce
```

Un costruttore è di solito **public**: se non lo fosse, i clienti non potrebbero creare oggetti della classe.

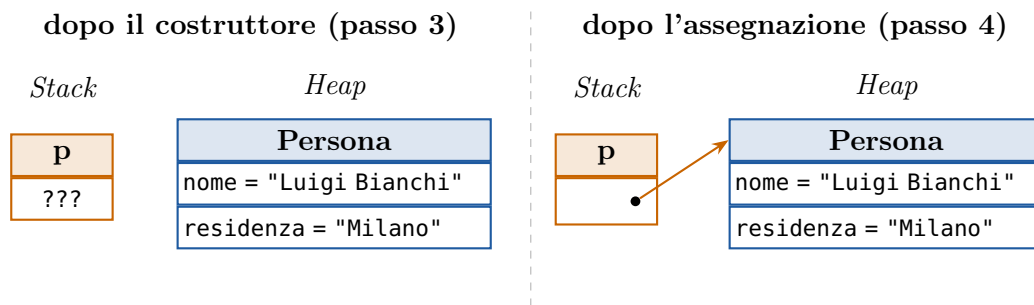
Nota – Il costruttore con **void**

Un metodo con il nome della classe e il tipo **void**, come **public void Persona(String n)**, compila, ma è un metodo qualsiasi e non un costruttore: **new** non lo invoca.

La creazione di un oggetto. **new** e il costruttore lavorano insieme. L'istruzione **p = new Persona("Luigi Bianchi", "Milano");** si svolge così:

1. la variabile **p** è nello stack, ancora senza valore;
2. **new** alloca nello heap l'oggetto, con i campi ai valori predefiniti: qui **null**;
3. **new** invoca il costruttore, che scrive i campi: l'oggetto di invocazione è quello appena allocato;
4. **new** restituisce il riferimento all'oggetto, che si assegna a **p**.

È **new**, gestito dalla JVM, a creare l'oggetto; il costruttore lo inizializza, e può farlo perché è codice della classe e accede ai campi privati.



*Il costruttore scrive i campi prima che **new** restituisca il riferimento; solo dopo **p** denota l'oggetto. Per semplicità le stringhe sono disegnate dentro i campi.*

Proposizione – Valori predefiniti dei campi dati

Un campo dati che il costruttore non scrive conserva il valore predefinito dato da **new**, lo stesso degli elementi di un array (Capitolo 5, Sezione 6): **0** per i numeri, **'\0'** per i **char**, **false** per i **boolean**, **null** per i riferimenti a oggetti.

Le variabili locali dei metodi invece non ricevono valori predefiniti, e vanno inizializzate (Capitolo 3, Sezione 8).

Overloading dei costruttori. Anche i costruttori ammettono l'overloading (Capitolo 5, Sezione 3): una classe può averne diversi, con parametri diversi, e **new** invoca quello che corrisponde ai parametri attuali.

```
public Persona(String n) {
    nome = n;
    residenza = null;
}
```

```
Persona p1 = new Persona("Luigi Bianchi");           // un parametro
Persona p2 = new Persona("Giovanni Verdi", "Roma"); // due parametri
```

Definizione – Costruttore standard

- Una classe senza costruttori ne riceve automaticamente uno, il **costruttore standard**: è senza parametri e lascia i campi ai valori predefiniti.
- Se la classe definisce un costruttore qualsiasi, il costruttore standard non viene generato: un costruttore senza parametri, se serve, va scritto.

Con la prima versione di **Persona**, senza costruttori, **new Persona()** invoca il costruttore standard. Con i due costruttori appena aggiunti la stessa istruzione non compila: *no suitable constructor found for Persona(no arguments)*. Il linguaggio evita così che un cliente crei, senza volerlo, una persona senza nome: chi ha scritto i costruttori ha deciso quali campi vanno sempre inizializzati.

Se serve, il costruttore senza parametri si aggiunge a mano. La classe completa:

```
public class Persona {  
    // campi dati (variabili di istanza)  
    private String nome;  
    private String residenza;  
  
    // costruttori  
    public Persona() {  
        nome = "Mario Rossi";  
        residenza = null;  
    }  
    public Persona(String n) {  
        nome = n;  
        residenza = null;  
    }  
    public Persona(String n, String r) {  
        nome = n;  
        residenza = r;  
    }  
  
    // campi funzione (metodi)  
    public String getNome() {  
        return nome;  
    }  
    public String getResidenza() {  
        return residenza;  
    }  
    public void setResidenza(String nuovaResidenza) {  
        residenza = nuovaResidenza;  
    }  
}
```

Non sempre un costruttore senza parametri ha senso: qui dà a ogni persona creata così un nome di comodo.

3 Realizzare una classe e i suoi clienti

Il problema: rappresentare automobili, di cui interessano la targa, il modello, il colore e il proprietario. Targa e modello non cambiano; colore e proprietario sì.

Definizione – Passi per realizzare una classe

1. Si sceglie la **rappresentazione** degli oggetti: i campi dati **private**, distinguendo le proprietà mutabili da quelle immutabili.
2. Si sceglie l'**interfaccia**: le intestazioni dei metodi **public**, costruttori compresi.
3. Si **realizzano** i metodi, scrivendone il corpo.

Scrivere subito il codice dei metodi, aggiungendo campi e cambiando gli accessi man mano che il compilatore protesta, porta a un progetto confuso: per esempio a rendere **public** un campo solo perché un cliente non compila.

Nota – Compilare spesso

Conviene compilare dopo ogni piccola aggiunta, a partire dalla classe vuota **public class Automobile { }**: un errore si trova subito, nelle ultime righe scritte. Chi compila per la prima volta dopo un'ora di lavoro si trova davanti decine di errori insieme.

Rappresentazione. Targa, modello e colore sono stringhe. Il proprietario è un oggetto della classe **Persona** della Sezione 2: un campo di un oggetto può contenere il riferimento a un altro oggetto.

Interfaccia. I costruttori inizializzano le proprietà immutabili:

- **Automobile(String t, String m, String c)**: anche il colore, perché un'auto-mobility nasce già verniciata; il proprietario resta **null**, come per le auto ancora in concessionaria;
- **Automobile(String t, String m, String c, Persona p)**: anche il proprietario, per le auto prodotte su ordinazione.

Poi un getter per ogni proprietà e un setter per il colore e per il proprietario.

Realizzazione.

```
public class Automobile {  
    // campi dati privati per la rappresentazione  
    private String targa;  
    private String modello;  
    private String colore;  
    private Persona proprietario;  
  
    // costruttori  
    public Automobile(String t, String m, String c) {  
        targa = t; modello = m; colore = c; proprietario = null;  
    }  
    public Automobile(String t, String m, String c, Persona p) {  
        targa = t; modello = m; colore = c; proprietario = p;  
    }  
  
    // funzionalità pubbliche  
    public String getTarga() { return targa; }  
    public String getModello() { return modello; }  
    public String getColore() { return colore; }  
    public void setColore(String nuovoc) { colore = nuovoc; }  
    public Persona getProprietario() { return proprietario; }  
    public void setProprietario(Persona nuovop) { proprietario = nuovop; }  
}
```

`proprietario = null` si potrebbe omettere, perché `null` è già il valore predefinito; scriverlo rende esplicito che l'auto nasce senza proprietario.

Un cliente con metodi statici. Verniciare un'automobile o immatricolarne una nuova sono servizi che non riguardano un oggetto in particolare, se non quello passato come parametro: si realizzano come metodi **static** di una classe cliente, che diventa un modulo funzionale (Capitolo 5, Sezione 5).

```
public class ServiziAuto {  
    public static void verniciatura(Automobile auto, String colore) {  
        auto.setColore(colore);  
    }  
    public static Automobile immatricolaFerrari360GT(String tar) {  
        return new Automobile(tar, "Ferrari 360 GT", "Rosso Maranello");  
    }  
}
```

Il metodo `verniciatura` cambia il colore dell'automobile ricevuta come parametro. Il metodo `immatricolaFerrari360GT` crea e restituisce un'automobile di modello e colore fissati. Il primo è un side effect sull'oggetto passato: il parametro `auto` denota lo stesso oggetto del parametro attuale (Capitolo 3, Sezione 10).

Una classe **Main** usa le tre classi:

```
public class Main {
    public static void main(String[] args) {
        Automobile a = new Automobile("313", "Cinquecento", "Rosso e Blu");
        stampaDatiAuto(a);
        Persona p = new Persona("Paperino", "Paperopoli");
        a.setProprietario(p);
        stampaDatiProprietario(a);
        ServiziAuto.verniciatura(a, "Rosso Maranello");
        stampaDatiAuto(a);
        Automobile b = ServiziAuto.immatricolaFerrari360GT("131");
        stampaDatiAuto(b);
        Persona c = new Persona("Clarabella", "Topolinia");
        b.setProprietario(c);
        stampaDatiProprietario(b);
    }

    // metodi ausiliari
    private static void stampaDatiAuto(Automobile a) {
        System.out.println("Automobile: " + a.getTarga() + ", "
            + a.getModello() + ", " + a.getColore());
    }
    private static void stampaDatiProprietario(Automobile a) {
        System.out.println("Proprietario: " + a.getProprietario().getNome()
            + ", " + a.getProprietario().getResidenza());
    }
}
```

```
Automobile: 313, Cinquecento, Rosso e Blu
Proprietario: Paperino, Paperopoli
Automobile: 313, Cinquecento, Rosso Maranello
Automobile: 131, Ferrari 360 GT, Rosso Maranello
Proprietario: Clarabella, Topolinia
```

L'espressione `a.getProprietario().getNome()` invoca `getNome` sull'oggetto restituito da `getProprietario`. I due metodi di stampa sono `private static`: ausiliari di `main`, che non servono ad altre classi.

Il programma ha quattro classi, **Main**, **Persona**, **Automobile** e **ServiziAuto**, ma è in esecuzione che lavorano gli oggetti: `main` li crea, li collega e invoca i primi metodi, e da lì la computazione procede con le invocazioni fra oggetti. Quanti oggetti crea è l'Esercizio 1.

4 Il riferimento **this**

Definizione – **this**

Nei metodi non **static** la parola chiave **this** denota l'**oggetto di invocazione**. Con **this.campo** e **this.metodo(...)** si accede ai campi e ai metodi di quell'oggetto; di solito si può omettere, come nei listati precedenti.

Un metodo **static** non ha oggetto di invocazione, quindi non può usare **this**: il compilatore segnala *non-static variable this cannot be referenced from a static context*.

Proposizione – Parametri con il nome di un campo

Un parametro con lo stesso nome di un campo lo **nasconde**: nel corpo del metodo il nome semplice denota il parametro. Il campo si raggiunge con **this**:

```
this.targa = targa;
```

assegna al campo **targa** il valore del parametro **targa**.

Così i parametri del costruttore possono avere i nomi dei campi, che li descrivono meglio di **t**, **m** e **c**:

```
public Automobile(String targa, String modello, String colore) {  
    this.targa = targa;  
    this.modello = modello;  
    this.colore = colore;  
    this.proprietario = null;  
}  
public void setColore(String colore) {  
    this.colore = colore;  
}
```

Senza **this**, **targa = targa**; compila ma assegna il parametro a se stesso: il campo resta **null**.

Riepilogo

- **Accesso:** campi dati `private`, servizi `public`; i campi `public` formano l'interfaccia pubblica.
- **Getter e setter:** il getter per ogni proprietà, il setter solo per le proprietà mutabili.
- **Costruttore:** nome della classe, nessun tipo restituito; lo invoca `new` dopo aver allocato l'oggetto.
- **Valori predefiniti:** i campi non scritti dal costruttore valgono `0`, `false` o `null`, come gli elementi di un array.
- **Overloading dei costruttori:** `new` sceglie il costruttore dai parametri attuali.
- **Costruttore standard:** senza parametri, generato solo se la classe non ne definisce altri.
- **Realizzazione:** rappresentazione, interfaccia, corpo dei metodi; i servizi senza oggetto in una classe cliente di metodi `static`.
- **this:** l'oggetto di invocazione; `this.campo = campo` quando il parametro ha il nome del campo.

5 Esercizi

Esercizio 1 – Quanti oggetti?

Il programma della Sezione 3 è formato dalle classi **Main**, **Persona**, **Automobile** e **ServiziAuto**. Quando **main** termina, quanti oggetti **Persona** e **Automobile** sono stati creati? Quali variabili di **main** li denotano, e quali campi li collegano fra loro?

Soluzione a pagina 13 →

Esercizio 2 – Punti e segmenti

Realizzare due classi Java:

1. **Punto**, per i punti dello spazio cartesiano, con le coordinate reali x , y , z , che non cambiano dopo la creazione. Senza coordinate, un punto nasce nell'origine.
2. **Segmento**, per i segmenti, dati dal punto iniziale e dal punto finale. Se si indica solo il punto finale, il segmento parte dall'origine.

Soluzione a pagina 13 →

6 Soluzioni

Soluzione dell'Esercizio 1 – Quanti oggetti?

Gli oggetti sono quattro: due **Automobile** e due **Persona**. Le classi sono definizioni; gli oggetti nascono solo con **new**.

Oggetto	Creato da	Variabile di main
Automobile 313	new in main	a
Persona Paperino	new in main	p
Automobile 131	new in immatricolaFerrari360GT	b
Persona Clarabella	new in main	c

Il campo **proprietario** dell'automobile 313 denota Paperino, quello della 131 Clarabella. **verniciatura** non crea oggetti: cambia il colore di quello denotato da **a**. Oltre a questi quattro esistono le stringhe, come i letterali e i risultati delle concatenazioni nelle stampe.

[← Torna all'Esercizio 1, pagina 12](#)

Soluzione dell'Esercizio 2 – Punti e segmenti

Le coordinate sono immutabili: campi **private** con il solo getter. Il costruttore a tre parametri usa **this**, perché i parametri hanno i nomi dei campi; quello senza parametri va scritto, perché il costruttore standard non viene generato.

```
public class Punto {
    private double x;
    private double y;
    private double z;

    public Punto(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    public Punto() {
        x = 0;
        y = 0;
        z = 0;
    }

    public double getX() { return x; }
    public double getY() { return y; }
```

```
public double getZ() { return z; }  
}
```

Un segmento ha due campi di tipo **Punto**. Il costruttore con il solo punto finale crea il punto iniziale con `new Punto()`, l'origine.

```
public class Segmento {  
    private Punto inizio;  
    private Punto fine;  
  
    public Segmento(Punto inizio, Punto fine) {  
        this.inizio = inizio;  
        this.fine = fine;  
    }  
    public Segmento(Punto fine) {  
        this.inizio = new Punto();  
        this.fine = fine;  
    }  
  
    public Punto getInizio() { return inizio; }  
    public Punto getFine() { return fine; }  
}
```

`x = 0` assegna l'intero `0` a un **double**, che vale `0.0`; i tre assegnamenti si potrebbero omettere, perché `0.0` è già il valore predefinito.

[*← Torna all'Esercizio 2, pagina 12*](#)