

Appunti di Progettazione del Software

Canale 1 · Prof. Paolo Liberatore

Capitolo 4: Incapsulamento e controllo dei dati

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Dati non validi e RuntimeException · private · Getter, setter e
incapsulamento

Oggetti non modificabili · Area con la formula di Erone · Campi che sono insiemi

Indice

1	Dati non validi	1
2	Il modificatore private	2
3	Incapsulamento: getter e setter	3
4	Oggetti non modificabili	6

1 Dati non validi

Proposizione – Controllo a esecuzione

Il compilatore controlla i tipi, non i valori. La validità dei dati non si può verificare in compilazione, in nessun linguaggio: i valori possono venire dall'utente, da un file, da un database o dalla rete, e si conoscono solo durante l'esecuzione. La classe li controlla quando li riceve, nel costruttore.

Con il `Triangolo` del Capitolo 3, `new Triangolo("vela", 1, 2, 4)` compila e si esegue, e il perimetro risulta 7; ma nessun triangolo ha quei lati. Allo stesso modo si crea un rettangolo di base -4 . Tre lunghezze sono i lati di un triangolo solo se ciascuna non supera la somma delle altre due (*proprietà triangolare*):

$$l_1 \leq l_2 + l_3, \quad l_2 \leq l_1 + l_3, \quad l_3 \leq l_1 + l_2.$$

Con l'uguaglianza il triangolo è schiacciato, di area zero. I lati 1, 2, 4 violano la terza: $4 > 1 + 2$.

Un commento accanto ai campi che spiega la regola non basta: chi usa la classe può non leggerlo, o inserire un dato sbagliato per errore.

Definizione – Eccezione

L'istruzione `throw new RuntimeException("messaggio");` genera un'eccezione: il programma si interrompe e la console mostra il messaggio, la classe e la riga dell'istruzione che l'ha generata e le chiamate che vi hanno portato.

```
Triangolo(String descrizione, int lato1, int lato2, int lato3) {
    super(descrizione);
    if (lato1 > lato2 + lato3
        || lato2 > lato1 + lato3
        || lato3 > lato1 + lato2) {
        throw new RuntimeException("Violazione della proprietà triangolare");
    }
    this.lato1 = lato1;
    this.lato2 = lato2;
    this.lato3 = lato3;
}
```

Il programma compila ancora. Quando l'esecuzione arriva a `new Triangolo("vela", 1, 2, 4)`, alla riga 3 di `Principale.java`, si ferma con:

```
Exception in thread "main" java.lang.RuntimeException: Violazione della
    proprietà triangolare
    at Triangolo.<init>(Triangolo.java:9)
    at Principale.main(Principale.java:3)
```

`Triangolo.<init>` è il costruttore. Il messaggio dice che cosa è successo, dove, e quale istruzione del `main` va corretta.

Osservazione – Segnalare subito

Un errore va segnalato il prima possibile e in modo evidente: un'eccezione al momento della creazione indica proprio l'istruzione da correggere. Interrompere il programma è però una scelta provvisoria: in un programma per gli utenti, un dato sbagliato inserito da tastiera si fa reinserire.

2 Il modificatore `private`

Definizione – `private`

Un campo o un metodo dichiarato **`private`** si può nominare solo dentro la propria classe, fra la graffa di apertura e quella di chiusura. Fuori, anche in una sottoclasse, scriverne il nome è un errore di compilazione.

Il controllo nel costruttore non impedisce di rendere sbagliato un triangolo creato giusto. Con i campi del Capitolo 3, senza modificatore, queste righe compilano e si eseguono senza errori:

```
Triangolo t = new Triangolo("vela", 3, 2, 4);
t.lato1 = 1;
System.out.println(t.perimetro());    // 7.0: i lati sono ora 1, 2, 4
```

Dichiarando i lati **`private int lato1`**; (e così gli altri due), la seconda riga non compila più. Il compilatore dà *`lato1 has private access in Triangolo`*, e lo stesso errore per `System.out.println(t.lato1)`: fuori dalla classe il nome non si può scrivere, né per assegnare né per leggere.

Che cosa conta.

- **Dove è scritto il nome.** Gli errori di accesso si scoprono solo in compilazione: non dipendono da chi esegue o da quali valori, ma da dove compare il nome nel programma.
- **Il campo dell'oggetto.** Dentro `Triangolo` si possono scrivere sia `this.lato1` sia, per un altro triangolo `altro`, `altro.lato1`. Un parametro o una variabile locale che si chiama `lato1` è un'altra variabile e non ha limiti.
- **Anche i metodi.** Un metodo **`private`** si chiama solo da dentro la classe: serve per i calcoli di appoggio.

Osservazione – Privato non vuol dire segreto

private non nasconde il valore: un metodo della classe può restituirlo, e chi lo chiama lo legge. Limita soltanto dove si può scrivere il nome del campo o del metodo.

Il contrario è **public**: un campo o un metodo **public** si può nominare in qualsiasi classe. In una classe possono convivere campi **private** e campi accessibili: **descrizione di Figura**, per esempio, può restare accessibile da fuori.

3 Incapsulamento: getter e setter

Definizione – Incapsulamento

Una classe è **incapsulata** se i suoi campi sono **private** e si leggono e si modificano solo attraverso metodi della classe:

- un **getter** (**getLato1**) restituisce il valore di un campo;
- un **setter** (**setLato1**) lo modifica, e può controllare il nuovo valore prima di assegnarlo.

Nomi e ordine. Il getter del campo **lato1** si chiama **getLato1**: **get** seguito dal nome del campo con l'iniziale maiuscola; il setter **setLato1**. Con nomi standard chi riceve la classe sa già come leggerne e modificarne i dati. Nella classe si scrivono prima i campi e il costruttore, poi getter e setter, poi gli altri metodi.

```
public class Triangolo extends Figura {
    private int lato1;
    private int lato2;
    private int lato3;

    Triangolo(String descrizione, int lato1, int lato2, int lato3) {
        super(descrizione);
        if (!latiValidi(lato1, lato2, lato3)) {
            throw new RuntimeException(
                "Violazione della proprietà triangolare");
        }
        this.lato1 = lato1;
        this.lato2 = lato2;
        this.lato3 = lato3;
    }

    private boolean latiValidi(int a, int b, int c) {
        return a <= b + c && b <= a + c && c <= a + b;
    }

    int getLato1() {
        return lato1;
    }

    void setLato1(int lato1) {
        if (!latiValidi(lato1, lato2, lato3)) {
            throw new RuntimeException(
                "Violazione della proprietà triangolare");
        }
        this.lato1 = lato1;
    }

    // perimetro() e toString() come nel Capitolo 3
}
```

Il setter controlla il nuovo lato insieme agli altri due e solo dopo lo assegna. Su un triangolo di lati 3, 2, 4, `t.setLato1(10)` lancia l'eccezione, e il triangolo resta com'era. Il controllo è scritto una volta sola, nel metodo privato `latiValidi`, che costruttore e setter usano.

Quando controllare. Il controllo si può fare in due momenti:

	Alla modifica	All'uso
dove	costruttore e setter	metodi che usano i dati (<code>perimetro</code> , <code>toString</code>)
oggetto non valido	mai	possibile, finché non lo si usa
errore	alla modifica sbagliata	al primo uso dei dati

Sono i due modi del Capitolo 2 per la coerenza dei collegamenti: impedire l'errore o rilevarlo. Per avere oggetti sempre validi si controlla alla modifica, e i campi devono essere **private**: altrimenti nessuno obbliga a passare dal setter.

Il quadrato. Con i campi del Capitolo 3, un quadrato si rompe con un assegnamento:

```
Quadrato q = new Quadrato("piastrella", 2);
q.altezza = 12;
System.out.println(q + " " + q.perimetro());
// Quadrato [descrizione=piastrella, lato=12] 28.0
```

Un quadrato di lato 12 dovrebbe avere perimetro 48: l'oggetto ha base 2 e altezza 12 e non è più un quadrato. Anche qui la soluzione è rendere **private** i campi di **Rettangolo**.

```
public class Rettangolo extends Figura {
    private int base;
    private int altezza;

    // costruttore, area() e perimetro() come nel Capitolo 3

    int getBase() {
        return base;
    }

    int getAltezza() {
        return altezza;
    }
}

// in Quadrato
public String toString() {
    return "Quadrato [descrizione=" + descrizione
        + ", lato=" + getAltezza() + "];"
}
```

Quadrato eredita `base` e `altezza`, e `super(descrizione, lato, lato)` li inizializza, ma non può nominarli: con `altezza` nel suo `toString` il compilatore dà *altezza has private access in Rettangolo*. Legge il lato con il getter. Senza setter, nessuno da fuori cambia base o altezza, e un quadrato resta un quadrato.

4 Oggetti non modificabili

Definizione – Oggetto non modificabile

Un oggetto è **non modificabile** se, una volta creato, i suoi dati non cambiano più. Si ottiene con campi **private** assegnati solo nel costruttore e nessun metodo che li modifichi.

Togliendo **setLato1** dalla classe della Sezione 3, l'unico punto che scrive i lati è il costruttore: il triangolo si crea con il controllo e poi non cambia. Restano i getter, e l'area può sostituire lo stub del Capitolo 3:

```

public class Triangolo extends Figura {
    private int lato1;
    private int lato2;
    private int lato3;

    // costruttore con il controllo, come nella Sezione 3

    int getLato1() {
        return lato1;
    }
    // getLato2() e getLato3() analoghi

    double perimetro() {
        return lato1 + lato2 + lato3;
    }

    double area() {
        double s = perimetro() / 2;    // semiperimetro
        return Math.sqrt(s * (s - lato1) * (s - lato2) * (s - lato3));
    }
}

```

`area()` usa la formula di Erone, $A = \sqrt{s(s-l_1)(s-l_2)(s-l_3)}$ con s il semiperimetro; per i lati 3, 2, 4 dà circa 2,90. Dentro la classe i campi privati si scrivono liberamente.

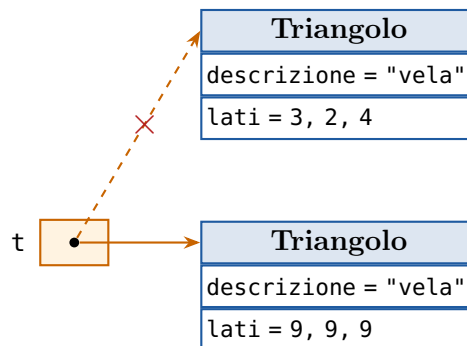
Anche **Rettangolo** e **Quadrato** della Sezione 3, che hanno solo getter, sono non modificabili. Lo è anche **String**: `s.charAt(i)` legge un carattere, ma nessun metodo lo cambia, e `s[3]` non compila (*array required, but String found*).

Osservazione – Un campo che è un insieme

Un getter non basta a proteggere un insieme. Se il campo `auto` della **Persona** del Capitolo 2 è `private` e la classe ha solo il getter `getAuto()`, l'istruzione `p.getAuto().add(a)` compila e aggiunge l'auto: il getter restituisce il riferimento all'insieme, e attraverso il riferimento l'insieme si modifica. Un vincolo come “da uno a cinque elementi” non si garantisce quindi con `private` e un getter soltanto.

Gli oggetti si creano solo con `new`, che chiama il costruttore, anche quando `new` sta dentro un metodo. Se il costruttore controlla i dati, nessun oggetto non valido esiste mai. Per “cambiare” un oggetto non modificabile se ne crea un altro:

```
Triangolo t = new Triangolo("vela", 3, 2, 4);
t = new Triangolo("vela", 9, 9, 9);
```



Dopo il secondo new: t contiene il riferimento al nuovo triangolo; il primo non è cambiato, e nessuna variabile lo raggiunge più.

Che cosa fa `t = new Triangolo("vela", 1, 2, 4);`. L'assegnamento valuta prima il lato destro.

1. `new` crea l'oggetto e chiama il costruttore.
2. Il costruttore trova i lati non validi e lancia l'eccezione.
3. L'assegnamento non avviene: `t` non riceve mai il riferimento a un triangolo non valido.

Riepilogo

- **Controllo a esecuzione:** la validità dei dati si verifica durante l'esecuzione, nella classe, a partire dal costruttore.
- **Eccezione:** `throw new RuntimeException("...")` interrompe il programma e indica messaggio, classe e riga.
- **private:** il nome si scrive solo fra le graffe della propria classe, nemmeno nelle sottoclassi; limita dove si scrive il nome, non chi conosce il valore.
- **Incapsulamento:** campi `private`, letti dai getter e modificati dai setter, che controllano il nuovo valore.
- **Quando controllare:** alla modifica (l'oggetto è sempre valido) oppure all'uso.
- **Non modificabile:** campi assegnati solo nel costruttore, nessun setter; come `String`.
- **Insiemi:** un getter che restituisce un insieme permette di modificarlo.
- **Assegnamento:** prima `new` e il costruttore, poi la variabile; se il costruttore lancia un'eccezione, la variabile non cambia.