

Appunti di Progettazione del Software

Canale 1 · Prof. Paolo Liberatore

Capitolo 2: Classi, associazioni e collezioni in Java

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Classi Persona e Automobile · Associazione e molteplicità in UML
Riferimenti e oggetti in memoria · Coerenza dei collegamenti
HashSet e liste · Garbage collection · Classi wrapper

Indice

1	Lo schema dei dati in UML	1
2	Le classi Java	2
3	Riferimenti e oggetti in memoria	5
4	Coerenza dei collegamenti	6
5	Insiemi e liste	8
6	Garbage collection	10
7	Tipi primitivi e classi wrapper	10
8	Esercizi	12
9	Soluzioni	13

1 Lo schema dei dati in UML

Osservazione – Un problema di struttura, non di algoritmo

L'esempio del capitolo è un piccolo gestionale di persone e automobili.

- Di una **persona** interessano nome, cognome e anno di nascita.
- Di un'**automobile** interessano targa e numero di telaio.
- Ogni persona possiede un insieme di automobili; ogni automobile ha un proprietario.

Non c'è un problema numerico da risolvere con un algoritmo, ma dati da rappresentare e collegare. Per questi collegamenti esistono soluzioni **standard**: in Java il codice che ne risulta è lungo, ma uguale a sé stesso qualunque siano i dati collegati (persone e automobili, studenti e corsi di laurea, ...).

Definizione – Classe, attributo, associazione in UML

Il **diagramma delle classi** di UML descrive quali dati interessano all'applicazione e come sono collegati.

- Una **classe** è un rettangolo con il nome in alto; modella un insieme di oggetti omogenei, le sue *istanze*.
- Nella parte inferiore si scrivono gli **attributi**, cioè i dati scalari della classe, con il loro tipo (**nome: stringa**).
- Un'**associazione** fra due classi C e D è una linea che le unisce: modella una relazione fra le loro istanze, cioè un sottoinsieme del prodotto cartesiano

$$R \subseteq C \times D.$$

Le sue istanze, le coppie di oggetti collegati, si chiamano **link**.

Nel diagramma si rappresentano solo i dati *di interesse per l'applicazione*: cilindrata, stato civile e simili restano fuori, se non servono.



L'insieme delle auto e il proprietario **non** compaiono fra gli attributi: li rappresenta la linea, e scriverli anche nei rettangoli sarebbe ridondante. La linea dice solo che persone e automobili possono essere collegate; quanti collegamenti ci possono essere lo dicono le molteplicità.

Definizione – Molteplicità

La **molteplicità** scritta a un'estremità dell'associazione indica a quante istanze di quella classe può essere collegata ogni istanza dell'altra classe.

Molteplicità	Significato
0..* (o *)	un numero qualsiasi; è il valore sottinteso se non si scrive nulla
0..1	al massimo una
1..1 (o 1)	esattamente una
1..*	almeno una
x..y	almeno x e al massimo y , con $0 \leq x \leq y$

Nel diagramma, l'1 dal lato di **Persona** dice che ogni automobile è posseduta da esattamente una persona; lo 0..* dal lato di **Automobile** dice che una persona può possedere da zero a un numero qualsiasi di automobili. Se un'automobile può anche non avere un proprietario, dal lato di **Persona** si scrive 0..1.

Osservazione – Lo schema non è il programma

Il diagramma descrive il *problema*: che esistono persone e automobili, quali dati interessano e come sono collegate. Non dice nulla su Java; lo si può disegnare prima di scrivere il codice, o senza implementarlo mai. A chi non conosce il progetto dice a colpo d'occhio su quali dati lavora il programma.

2 Le classi Java

Nota – Dal diagramma delle classi al codice

Nel diagramma	In Java
classe	classe con lo stesso nome
attributo di tipo <code>stringa</code>	campo <code>String</code>
attributo di tipo <code>int</code>	campo <code>int</code>
molteplicità 0..* dal lato di <i>Altra</i>	campo <code>HashSet<Altra></code>
molteplicità 1 o 0..1 dal lato di <i>Altra</i>	campo di tipo <i>Altra</i> , null se manca

Il campo che rappresenta l'associazione sta nella classe all'altro estremo della linea: l'insieme **auto** in **Persona**, creato vuoto nel costruttore, e il campo **proprietario** in **Automobile**.

Secondo i massimi delle due molteplicità, un'associazione è di uno di tre tipi; con la tabella, ogni classe ha un campo singolo se dall'altro lato il massimo è 1, un insieme altrimenti.

Tipo	Esempio	Campi
uno a uno	cliente e indirizzo	un campo singolo in ciascuna classe
uno a molti	persona e automobili	un insieme da un lato, un campo dall'altro
molti a molti	clienti e conti cointestati	un insieme in ciascuna classe

Ogni collezione Java è una classe di libreria: va **importata** (`HashSet` sta nel pacchetto `java.util`) e va **dichiarata** con il tipo degli elementi.

```
import java.util.HashSet;

public class Persona {
    String nome;
    String cognome;
    int nascita;
    HashSet<Automobile> auto;    // le automobili possedute

    Persona(String nome, String cognome, int nascita) {
        this.nome = nome;
        this.cognome = cognome;
        this.nascita = nascita;
        this.auto = new HashSet<Automobile>();    // insieme vuoto
    }
}
```

```
public class Automobile {
    String targa;
    String telaio;
    Persona proprietario;    // null finché non viene assegnato

    Automobile(String targa, String telaio) {
        this.targa = targa;
        this.telaio = telaio;
    }
}
```

Definizione – Costruttore

Il **costruttore** è il metodo che ha lo stesso nome della classe e nessun tipo di ritorno (neanche `void`). Inizializza i campi dell'oggetto nel momento in cui nasce, e si invoca solo con `new`.

Nota – Costruttore di default

Una classe che non dichiara costruttori ha il **costruttore di default**, senza argomenti, che Java aggiunge da sé e che lascia i campi ai valori iniziali. Se la classe dichiara un costruttore, quello di default non c'è più: con la classe **Persona** qui sopra, `new Persona()` è un errore e si deve scrivere `new Persona("Mario", "Rossi", 1980)`.

Nota – Dichiarare non basta

La riga `HashSet<Automobile> auto;` dichiara soltanto la variabile. L'insieme vero e proprio, inizialmente vuoto, va creato con `new HashSet<Automobile>()`, qui nel costruttore. Senza questa riga il campo `auto` vale `null` e il primo `p1.auto.add(a)` fa fallire il programma.

Il proprietario non passa per il costruttore: un'automobile nasce senza proprietario, che si assegna dopo, quindi il codice realizza la molteplicità 0..1.

Un elemento si aggiunge a un insieme con il metodo `add`. Aggiungere un'auto alla persona però non basta: bisogna anche registrare la persona come proprietario dell'auto.

```
public class Principale {
    public static void main(String[] args) {
        Persona p1;
        p1 = new Persona("Mario", "Rossi", 1980);
        Automobile a;
        a = new Automobile("AB123CD", "ZFA1990000123456");

        p1.auto.add(a);           // la persona possiede l'auto...
        a.proprietario = p1;     // ...e l'auto ha la persona come proprietario

        for (Automobile x : p1.auto) {
            System.out.println(p1.nome + " possiede " + x.targa);
        }
        System.out.println(a.targa + " appartiene a " + a.proprietario.nome);
    }
}
```

Stampa `Mario possiede AB123CD` e `AB123CD appartiene a Mario`: dalla persona si arriva alle sue auto e dall'auto al proprietario. Il ciclo `for` che scorre l'insieme è il for-each della Sezione 5.

3 Riferimenti e oggetti in memoria

Definizione – Variabili riferimento

In Java una variabile di tipo classe non contiene l'oggetto, ma un **riferimento** all'oggetto, cioè un modo per raggiungerlo (in pratica un indirizzo). Gli oggetti sono creati in memoria indipendentemente dalle variabili. Fanno eccezione i **tipi primitivi** (`int`, `double`, `char`, `boolean`, ...), che la variabile contiene direttamente.

In C si sceglie fra una variabile `struct` e una variabile `struct *`; in Java le variabili di tipo classe sono sempre riferimenti, e il linguaggio li gestisce da sé.

Che cosa fa new. L'istruzione `p1 = new Persona(...)`; si esegue come ogni assegnazione: prima la parte a destra, poi il risultato va nella variabile a sinistra.

1. `new` crea in memoria l'oggetto `Persona`, con i suoi campi, ed esegue il costruttore: è l'analogo della `malloc` del C;
2. l'indirizzo dell'oggetto appena creato (diciamo 1000) viene scritto in `p1`.

In C l'indirizzo si potrebbe stampare; in Java no, e basta sapere che `p1` è un riferimento all'oggetto.

Nota – Campi e variabili locali

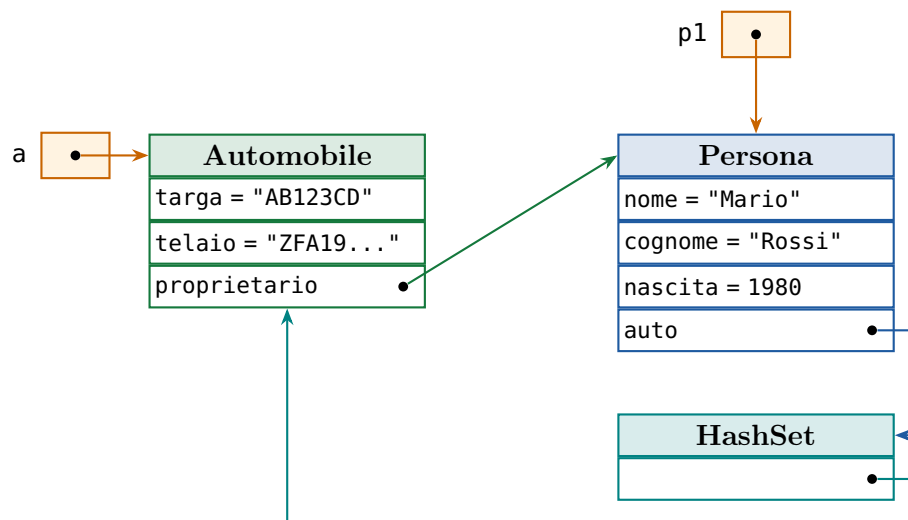
Prima dell'assegnazione, che cosa contiene `p1`? Dipende da dove è dichiarata la variabile.

- I **campi** di un oggetto sono inizializzati in modo implicito: `0` per `int`, `false` per `boolean`, `null` per i riferimenti. Per questo **proprietario** vale `null` finché non lo si assegna.
- Le **variabili locali** di un metodo, come `p1` e `a` nel `main`, non sono inizializzate: il compilatore rifiuta il programma se le si usa prima di aver dato loro un valore (Esercizio 1).

Il valore speciale `null` indica un riferimento che non punta ad alcun oggetto.

Anche l'insieme è un oggetto. Il campo `auto` è a sua volta un riferimento, a un oggetto `HashSet`, di cui non interessa la struttura interna. L'insieme contiene zero o più **riferimenti** ad automobili, non gli oggetti `Automobile` stessi.

Il campo `proprietario` dell'automobile contiene un riferimento allo stesso oggetto a cui punta `p1`: due riferimenti diversi portano allo stesso oggetto. Dopo il `main` della Sezione 2 la memoria è questa.



Variabili (arancione) e oggetti in memoria: le frecce sono riferimenti. L'insieme contiene un riferimento all'automobile, l'automobile uno alla persona.

Nota – Collegamenti diretti e basi di dati

In una base di dati **relazionale** ogni classe è una tabella e ogni oggetto una riga; anche ogni associazione è una tabella, che elenca le coppie collegate. In Java i collegamenti sono invece **diretti**: la persona ha i riferimenti alle sue auto, l'auto quello al proprietario, come nelle basi di dati **reticolari**. Le prime si usano per conservare i dati su disco, la rappresentazione con riferimenti è quella comune nella memoria di un programma in esecuzione.

4 Coerenza dei collegamenti

Il collegamento fra una persona e un'automobile è memorizzato **due volte**: nel campo **proprietario** dell'auto e nell'insieme **auto** della persona. Le due informazioni devono essere **coerenti**: se il proprietario di **a** è **p1**, allora **a** deve stare fra le auto di **p1**, e viceversa.

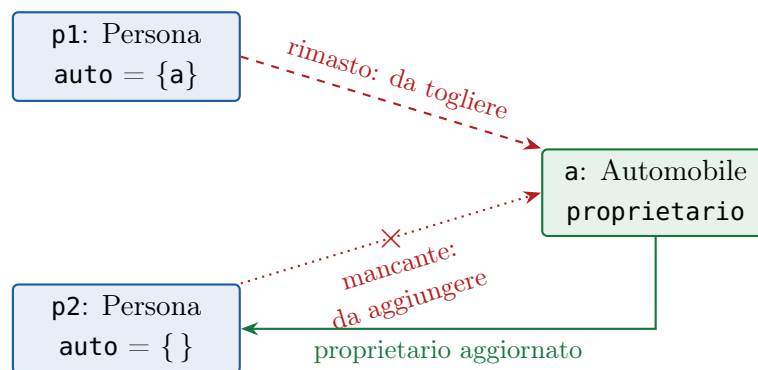
Esempio – Un cambio di proprietario fatto a metà

Chi ha scritto le classi pensava che un'automobile non cambiasse mai proprietario. Chi in seguito deve gestire le vendite scrive:

```
Persona p2 = new Persona("Anna", "Bianchi", 1992);
a.proprietario = p2;
System.out.println(p1.auto.contains(a)); // true
System.out.println(p2.auto.contains(a)); // false
```

Il programma stampa **true** e **false**: il collegamento è stato aggiornato in un verso solo, e le inconsistenze sono due.

1. Il proprietario di **a** è **p2**, ma **a** non è fra le auto di **p2**.
2. **a** non ha più **p1** come proprietario, ma è ancora fra le auto di **p1**.



Dopo `a.proprietario = p2`; i due versi del collegamento non descrivono più la stessa situazione.

Per cambiare proprietario servono tre operazioni:

1. togliere **a** dalle auto di **p1**;
2. aggiungere **a** alle auto di **p2**;
3. aggiornare **a.proprietario**.

Basta dimenticarne una perché nascano una o due inconsistenze, a seconda di quale. L'errore capita soprattutto quando sul codice lavorano più programmatori, in tempi diversi.

Collegamenti in un verso solo. Il problema esiste solo quando il collegamento è rappresentato in *entrambi* i versi. Se ogni persona è collegata alla propria nazione, dalla persona si risale alla nazione; collegare ogni nazione a tutti i suoi abitanti vorrebbe dire, per l'Italia, circa 60 milioni di collegamenti. Lì basta il verso persona → nazione, e senza collegamento inverso non c'è coerenza da garantire.

Per persone e automobili servono invece tutti e due i versi. Senza l'insieme **auto**, per stampare le auto di una persona si dovrebbero scorrere tutte le automobili create e tenere quelle con quel proprietario: con migliaia di automobili, a ogni richiesta.

Due modi di garantire la coerenza.

1. **Impedire l'errore:** i due collegamenti si possono creare o modificare solo insieme, dentro un unico metodo che, quando aggiorna uno, aggiorna anche l'altro.
2. **Rilevare l'errore:** i collegamenti si possono aggiornare liberamente, anche in modo sbagliato, ma l'eventuale incoerenza viene segnalata quando si chiedono le auto di una persona o il proprietario di un'auto.

Il primo modo, fatto in modo da rendere l'errore davvero impossibile, richiede un meccanismo complicato; nel corso si usa il secondo, più semplice. Entrambi richiedono di limitare l'accesso ai campi con i modificatori **private** e **public**: finché i campi sono accessibili da qualsiasi classe, chiunque può scrivere `a.proprietario = p2;`. Senza questi modificatori l'unica garanzia è che tutti sappiano che i due collegamenti vanno aggiornati insieme.

5 Insiemi e liste

Definizione – HashSet e LinkedList

- Un **insieme** (**Set**, realizzato per esempio da **HashSet**, basato su una tavola hash) contiene elementi **senza ordine**: non c'è un primo né un secondo elemento.
- Una **lista** (**List**, realizzata per esempio da **LinkedList**, una lista collegata) è una **sequenza**: ogni elemento ha un indice, da **0** in su, come negli array del C. Gli elementi aggiunti con **add(e)** vanno in coda.

Entrambe le classi sono in `java.util`.

```
HashSet<String> h = new HashSet<String>();  
h.add("rosso");  
h.add("verde");  
h.add("arancione");  
  
LinkedList<String> l = new LinkedList<String>();  
l.add("primo");  
l.add("secondo");  
l.add("terzo");  
  
System.out.println(l.get(1));    // secondo
```

Accesso per indice. Viene spontaneo scrivere `l[1]`, ma il compilatore lo segnala come errore: `l` è un oggetto, non un array. L'elemento di indice `1`, cioè il *secondo*, si ottiene con il metodo `l.get(1)`. Per un insieme non esiste niente di simile, perché non esiste un secondo elemento.

Inserire in mezzo a una lista. Una lista, a differenza di un array, permette di inserire un elemento in qualsiasi posizione con `add(indice, elemento)`: l'elemento nuovo prende quell'indice e quelli che seguono scalano di un posto.

```
l.add(1, "nuovo");  
System.out.println(l);    // [primo, nuovo, secondo, terzo]
```

L'ordine di un insieme non è garantito. A volte la stampa di un insieme sembra rispettare l'ordine di inserimento, ma il programma non deve contarci. Con l'insieme `h` qui sopra l'ordine è già diverso:

```
for (String s : h) {  
    System.out.println(s + " " + s.length());  
}
```

stampa rosso 5, arancione 9, verde 5, benché "verde" sia stato inserito prima di "arancione".

Definizione – Ciclo for-each

Il ciclo `for (T x : c) { ... }` esegue il corpo una volta per ogni elemento della collezione `c`, con `x` uguale a quell'elemento. È il modo per visitare gli elementi di un insieme, e si usa comodamente anche con le liste e gli array; corrisponde al `for x in c` di Python.

Rispetto a stampare direttamente la collezione, il ciclo permette di fare con ogni elemento quello che si vuole: nell'esempio si stampano la stringa e la sua lunghezza, `s.length()`. L'Esercizio 1 lo usa per cercare la stringa più lunga.

6 Garbage collection

Definizione – Garbage collection

La **garbage collection** è il recupero della memoria occupata dagli oggetti che il programma non può più raggiungere, perché nessun riferimento porta a loro. In Java la svolge automaticamente la macchina virtuale (JVM), con il *garbage collector* (GC).

In C la memoria ottenuta con `malloc` si libera con `free`; in Java non c'è nessuna istruzione per liberare un oggetto.

```
Automobile b = new Automobile("CD456EF", "WVW2000000654321");
b = new Automobile("GH789LM", "VF32000000987654");
// la prima automobile non è più raggiungibile: il GC può recuperarla
```

7 Tipi primitivi e classi wrapper

Definizione – Classe wrapper

Per ogni tipo primitivo esiste una **classe wrapper**, i cui oggetti contengono un valore di quel tipo:

<code>byte</code>	<code>short</code>	<code>int</code>	<code>long</code>	<code>float</code>	<code>double</code>	<code>char</code>	<code>boolean</code>
<code>Byte</code>	<code>Short</code>	<code>Integer</code>	<code>Long</code>	<code>Float</code>	<code>Double</code>	<code>Character</code>	<code>Boolean</code>

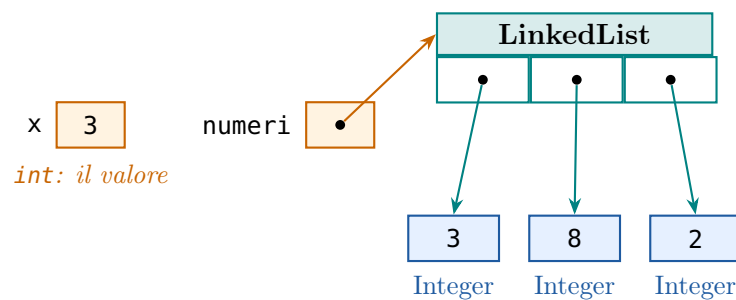
Le classi wrapper fanno parte di `java.lang` e non vanno importate.

Servono perché le collezioni contengono riferimenti a oggetti, mentre una variabile `int` contiene direttamente il numero. `LinkedList<int>` non si può scrivere: il compilatore dà l'errore *unexpected type*, perché si aspetta un tipo riferimento.

```

LinkedList<Integer> numeri = new LinkedList<Integer>();
numeri.add(3);
numeri.add(8);
numeri.add(2);
int somma = 0;
for (Integer n : numeri) {
    somma = somma + n;
}
System.out.println(numeri + " " + somma);    // [3, 8, 2] 13
    
```

Quando si scrive `numeri.add(3)`, Java prende il 3, lo mette in un nuovo oggetto `Integer` e aggiunge alla lista il riferimento a quell'oggetto; lo stesso per 8 e 2. Questa conversione automatica si chiama **autoboxing**; quella inversa, da `Integer` a `int` come in `somma + n`, **unboxing**.



Una variabile `int` contiene il numero; la lista contiene riferimenti a oggetti `Integer`.

Riepilogo

- Il **diagramma delle classi** UML descrive i dati del problema: classi con attributi scalari, associazioni come linee, molteplicità agli estremi (1, 0..1, 0..*).
- In Java le variabili di tipo classe sono **riferimenti**; `new` crea l'oggetto e ne restituisce il riferimento. Anche le collezioni sono oggetti e contengono riferimenti.
- Un'associazione navigabile nei due versi si memorizza con due collegamenti, che vanno **aggiornati insieme**; se basta un verso, il problema della coerenza non si pone.
- `HashSet`: nessun ordine, nessun indice. `LinkedList`: indici da 0, `get(i)`, `add(i, e)`. Entrambe si visitano con il **for-each**.
- Le variabili locali vanno inizializzate prima dell'uso; i campi hanno valori predefiniti (`null` per i riferimenti).
- Non esiste **free**: gli oggetti non più raggiungibili li recupera il **garbage collector**.
- I tipi primitivi entrano nelle collezioni attraverso le **classi wrapper** (`Integer`, `Character`, ...), con autoboxing e unboxing.

8 Esercizi

Esercizio 1 – La stringa più lunga

Scrivere un metodo che, data una collezione di stringhe (un insieme o una lista), restituisca la stringa più lunga: la stringa, non la sua lunghezza.

Soluzione a pagina 13 →

9 Soluzioni

Soluzione dell'Esercizio 1 – La stringa più lunga

Si scorre la collezione tenendo la stringa più lunga trovata finora; quando se ne trova una più lunga, la si sostituisce.

```
import java.util.Collection;

public class Collezioni {
    static String piuLunga(Collection<String> c) {
        String max = "";
        for (String s : c) {
            if (s.length() > max.length()) {
                max = s;
            }
        }
        return max;
    }
}
```

`Collection<String>` è il tipo comune a insiemi e liste: il parametro accetta sia un `HashSet<String>` sia una `LinkedList<String>`. Il metodo è `static`, così si chiama direttamente dal `main` senza creare un oggetto.

Con l'insieme `h` e la lista `l` della Sezione 5, `piuLunga(h)` restituisce "arancione" e `piuLunga(l)` restituisce "secondo". Su una collezione vuota restituisce la stringa vuota.

Errore tipico. Con solo `String max;` il compilatore dà due volte l'errore *variable max might not have been initialized*: prima su `max.length()` nell'`if`, poi su `return max;`. Al primo confronto `max` non ha ancora un valore, e con una collezione vuota non lo riceve mai. Per questo la si inizializza con la stringa vuota.

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