Il “Collections Framework”

Sezioni

Questa sezione presenta in modo molto succinto alcune interfacce e classi del “Collections Framework” con particolare attenzione agli usi delle medesime che possono risultare utili per la prova pratica.

Come per il caso delle interfacce di comparazione, una discussione esaustiva di questo argomento esula dagli scopi di questo documento, chi volesse approfondire è invitato a iniziare la sua esplorazione dalla documentazione nel JDK e il (pur datato) tutorial ufficiale.

Le *collezioni* che possono rivelarsi utili all’esame sono organizzate in due famiglie

- i sottotipi dell’interfaccia `Collection`, limitatamente a quelli di `List`, `Set` e `SortedSet`;
- i sottotipi delle interfacce `Map` e `SortedMap`.

Osservate che in ossequio alla consuetudine comune, come si evince da quanto sopra, in seguito chiameremo *collezioni* non solo i sottotipi di `Collection`, ma anche quelli di `Map`.

Per ciascuna di queste interfacce, le API contano diverse implementazioni che si distinguono tra loro per vari aspetti, per quel che concerne la prova d’esame la differenza principale riguarda l’efficienza rispetto a determinate operazioni.

Le versioni *ordinate* delle interfacce (quelle il cui nome inizia per `Sorted`), dipendono dall’*ordine naturale* degli elementi (ossia dal fatto che siano *comparabili*), oppure da un eventuale altro loro ordinamento, specificato tramite un *comparatore* alla costruzione della collezione.

Per le *liste* le due implementazioni maggiormente utili sono

- `ArrayList` che è basata su un array e perciò consente un efficiente *accesso causale*, e
- `LinkedList` che è basata sulle *liste doppiamente concatenate* e perciò consente efficienti operazioni di inserimento e cancellazione.

Per *insiemi* e *mappe* le implementazioni maggiormente utili sono

- `HashSet` e `HashMap`, basate su *hash table*;
- `TreeSet` e `TreeMap`, basate su *alberi rosso-nero*;

evidentemente le implementazioni basate su *hash table* non sono ordinate e richiedono che i metodi `equals` e `hashCode` degli elementi siano opportunamente sovrascritti, similmente le implementazioni basate su *alberi rosso-nero* sono ordinate e richiedono che gli elementi siano ordinati (nel senso di cui sopra).

Si rimanda alle conoscenze acquisite dall’insegnamento di “Algoritmi e strutture dati” per le questioni inerenti l’efficienza delle varie operazioni a seconda delle implementazioni scelte.

Immutabilità e viste

Iniziamo con alcune considerazioni di carattere generale, che pongono in relazione le collezioni con una delle nozioni centrali dell'insegnamento: l'*immutabilità*.

Collezioni non modificabili

Una collezione è *immutabile* se:

- gli elementi che contiene sono a loro volta *immutabili* e
- non può essere *strutturalmente modificata*, ossia non possono essere aggiunti, eliminati, o riordinati i suoi elementi.

Riguardo al primo punto, è responsabilità del progettista del tipo degli elementi decidere se e come renderli immutabili (o se e come “proteggerli”, quando le collezioni entrano a far parte della rappresentazione di un oggetto).

Riguardo al secondo punto, viceversa, esistono varie implementazioni delle collezioni che garantiscono la *non modificabilità*; dato che evidentemente non è possibile che non implementino i metodi *mutazionali* presenti nelle interfacce, la soluzione adottata è che essi, qualora invocati, sollevino l'eccezione non controllata `UnsupportedOperationException`.

Alcune implementazioni non modificabili sono ottenute tramite una *vista non modificabile*, ragion per cui nella prossima sezione sarà illustrato il concetto generale di *vista di una collezione*.

Viste

La *vista di una collezione* è una implementazione di una collezione che invece di gestire direttamente la memorizzazione dei suoi elementi fa uso di una collezione (o array) di appoggio per immagazzinarli concretamente; le operazioni che non possono essere implementate direttamente dalla vista sono delegate alla collezione (o array) d'appoggio.

Le viste non occupano alcuno spazio per memorizzare gli elementi (nemmeno quello dei riferimenti), quindi hanno un basso costo in termini di spazio; d'altro canto, la necessità di delegare molti comportamenti alla collezione (o array) d'appoggio determina un piccolo costo in termini di tempo.

È di *fondamentale importanza* ricordare però che, per come sono costruite, i cambiamenti delle collezioni d'appoggio si riflettono però sempre nella viste!

Un caso tipico di vista sono le *sottocollezioni*, come ad esempio le *sottoliste* che possono essere ottenute tramite il metodo `subList`

```
List<Integer> lista = new ArrayList<>(List.of(1, 2, 3, 5, 6));
List<Integer> sottolista = lista.subList(2, 4);
lista + "; " + sottolista
```

```
[1, 2, 3, 5, 6]; [3, 5]
```

occorre prestare sempre attenzione a come le alterazioni *anche strutturali* della sottocollezione alterano la collezione; ad esempio

```
sottolista.add(1, 4);
lista + "; " + sottolista
```

```
[1, 2, 3, 4, 5, 6]; [3, 4, 5]
```

Copie e viste non modificabili

Ci sono diversi modi di ottenere una collezione non modificabile:

- fabbricandone una ex-novo a partire da un elenco di elementi (o coppie chiave e valore) passati come argomento a un opportuno metodo statico, oppure
- fabbricando una *copia* o una *vista* non modificabili di una collezione esistente.

Ogni interfaccia contiene una serie di metodi statici `of` (di arietà crescente, fino a quello variadico) per fabbricare una collezione del suo tipo; ad esempio

```
List<String> lista = List.of("uno", "due", "due");
  Set<String> insieme = Set.of("uno", "due", "tre");
Map<String, Integer> mappa = Map.of("uno", 1, "due", 2, "tre", 3);
lista + "; " + insieme + "; " + mappa
```

```
[uno, due, due]; [due, tre, uno]; {due=2, tre=3, uno=1}
```

nel caso delle mappe c’è anche il metodo statico **`ofEntries`** che può essere comodamente usato importando staticamente `java.util.Map.entry`

```
import static java.util.Map.entry;

Map<String, Integer> altraMappa = Map.ofEntries(
  entry("quattro", 4),
  entry("cinque", 5),
  entry("sei", 6)
);
altraMappa
```

```
{cinque=5, sei=6, quattro=4}
```

Ogni interfaccia contiene inoltre i metodi `copyOf` per fabbricare una collezione copiando i riferimenti agli elementi dalla collezione passata come argomento (similmente al caso dell’omonimo metodo della classe `Arrays` — anche in questo caso, non vengono copiati gli elementi, ma solo i loro riferimenti); ad esempio

```
Set<String> copia = Set.copyOf(insieme);
copia
```

```
[due, tre, uno]
```

> **Suggerimento**
>
> Le collezioni fabbricate con `of` e `copyOf` non possono contenere `null` nel senso che se esiste un tale valore tra gli elementi della collezione da copiare, verrà sollevata una `NullPointerException`; questo può essere molto comodo quando si vuole assegnare ad un attributo di una classe una collezione che sia non nulla e non contenga elementi nulli; se poi gli elementi sono immutabili, ciò basta per garantire l’immutabilità della copia!

Per finire, la classe di metodi statici di utilità **`Collections`** (che incontreremo di nuovo in seguito), contiene i metodi per fabbricare viste non modificabili delle varie collezioni, essi hanno nome `unmodifiableT` dove `T` è uno delle possibili interfacce per le collezioni; ad esempio **`unmodifiableSet`** consente di ottenere un insieme non modificabile

```
Set<String> mutabile = new HashSet<>();
mutabile.addAll(List.of("primo", "secondo", "terzo"));
Set<String> immutabile = Collections.unmodifiableSet(mutabile);
try {
  immutabile.add("nuovo");
} catch (UnsupportedOperationException e) {
  System.err.println("Modifica non consentita!");
}
```

```
Modifica non consentita!
```

mostra come l’invocazione di un metodo mutazionale sulla vista sollevi in effetti l’eccezione attesa; attenzione però: come già detto parlando delle viste, se la collezione sottostante cambia, la modifica si riflette necessariamente anche nella vista

```
mutabile.add("ultimo");
immutabile
```

```
[ultimo, terzo, primo, secondo]
```

> **Suggerimento**
>
> Le viste non modificabili possono essere molto utili nel caso uno degli attributi di una classe sia una collezione modificabile e si intenda rendere la classe un *iterabile* degli elementi di tale collezione. Restituire direttamente l’iteratore ottenuto dalla collezione potrebbe *esporre la rappresentazione* della classe conentendone la modifica dall’esterno (alcuni iteratori implementano infatti il metodo `remove` che consentirebbe di elimianre gli elementi della collezione durante l’iterazione); ciò è facilmente evitabile restituendo invece l’iteratore della vista non modifdicabile. Ad esempio
>
> ```
> class AClass implements Iterable<AType> {
>   private final Collection<AType> aModifiableCollection;
>
>   // qui i costruttori e altri metodi di AClass
>
>   @Override
>   public Iterator<AType> iterator() {
>     return Collections.unmodifiableCollection(aModifiableCollection).iterator();
>   }
> }
> ```

Copie modificabili

Talvolta può essere utile costruire una collezione modificabile a partire dagli elementi contenuti in un’altra collezione. Ci sono due modi molto pratici per ottenere una tale collezione:

- costruirla tramite un *costruttore copia*,
- costruire una collezione vuota ed aggiungergli tutti gli elementi di quella esistente,
- invocare il metodo `clone`.

Ogni collezione ha un costruttore copia che prende una `Collection` per argomento e costruisce una nuova collezione che contiene un nuovo riferimento per ciascun elemento (ma non dell’elemento) della collezione da cui è copiata; ad esempio

```
List<String> listaModificabile = new ArrayList<>(List.of("uno", "due", "tre"));
List<String> copia = new LinkedList(listaModificabile);
listaModificabile.add("quattro");
copia.remove("tre");
listaModificabile + "; " + copia
```

```
[uno, due, tre, quattro]; [uno, due]
```

una diversa strategia è quella di usare il metodo `addAll`, come la precedente può essere usata anche nel caso in cui la destinazione sia di tipo diverso dalla sorgente della copia; ad esempio

```
SortedSet<String> vuoto = new TreeSet<>();
vuoto.addAll(listaModificabile);
vuoto
```

```
[due, quattro, tre, uno]
```

Come è facile dedurre (anche osservando gli esempi), i costruttori copia possono essere usati anche per “convertire” il tipo di una collezione (ad esempio da *lista* a *insieme*, oppure tra implementazioni diverse dello stesso tipo).

Riguardo al metodo `clone`, esso restituisce un `Object` in ottemperanza al contratto che eredita da `Object`, per qui il suo uso richiede un cast; per questa ragione è generalmente preferibile l’uso dei costruttori copia.

Array e `Collection`

Come è ovvio attendersi, c’è un notevole legame tra array e collezioni.

Da array a liste

In un verso, la classe `Arrays` ha il metodo variadico **`asList`** che può essere usato per costruire una lista a partire da un array di riferimenti (ossia non di tipi primitivi); tale lista si comporta come una vista non modificabile. Ad esempio

```
String[] mksUnits = new String[] {"metro", "kilo", "secondo"};
List<String> comeLista = Arrays.asList(mksUnits);
comeLista
```

```
[metro, kilo, secondo]
```

Attenzione che, come accade nelle viste, se cambia l’array allora cambia la lista

```
mksUnits[1] = "kilogrammi";
comeLista.get(1)
```

```
kilogrammi
```

Si osservi per inciso che l’uso delle viste costruite a partire da un array offre la possibilità di effettuare in modo conveniente tramite il metodo **`indexOf`** (o lastIndexOf che inizia la ricerca dal fondo e quindi darà risultati diversi in caso di elementi ripetuti) una *ricerca sequenziale* tra i suoi elementi (basata sul loro metodo `equals`).

Rivisitando l’esempio delle cifre, l’inversa della funzione che mappa `i` in `cifreInParole[i]`, ossia la funzione `cifraAValore` che mappa la parola `cifra` corrispondente a una cifra in parole nel suo valore `i` è data da

```
static final int cifraAValore(final String cifra) {
  int valore = Arrays.asList(cifreInParole).indexOf(cifra);
  if (valore < 0) throw new IllegalArgumentException();
  return valore;
}
```

che (certamente ad un costo lineare) è però in grado di convertire parole in valori

```
cifraAValore("due")
```

```
2
```

Occorre prestare però molta attenzione al metodo `asList` nel caso di argomenti che siano di tipo primitivo, sopratutto array con elementi di tipo primitivo! È evidente che, non potendo istanziare tipi generici con tipi primitivi, l’invocazione di

```
List<Integer> listaDiInteger = Arrays.asList(1, 2, 3);
listaDiInteger
```

```
[1, 2, 3]
```

non può che restituire una lista di `Integer`. Osservando che un metodo variadico può essere equivalentemente invocato oltre che con un elenco di argomenti con un array di tali elementi, è anche del tutto atteso che

```
Integer[] arrayDiInteger = new Integer[] {4, 5, 6};
Arrays.asList(arrayDiInteger)
```

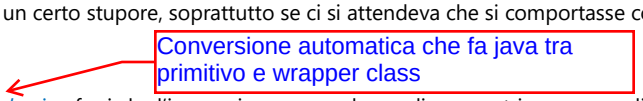
```
[4, 5, 6]
```

produca lo stesso risultato del codice precedente. Il risultato del seguente codice

```
int[] arrayDiInt = new int[] {4, 5, 6};
Arrays.asList(arrayDiInt)
```

```
[[I@772f98c3]
```

non può però che destare un certo stupore, soprattutto se ci si attendeva che si comportasse come i casi precedenti. Quel che accade, è che



- nel primo caso, l'*autoboxing* fa si che l'invocazione su un elenco di parametri `int` venga di fatto indirizzata al metodo in cui il parametro di tipo corrisponde ad `Integer` (di arietà 3);
- nel secondo caso, l'array di tipo `Integer[]` gioca esattamente lo stesso ruolo dell'elenco di argomenti (e l'invocazione è indirizzata alla segnatura di arietà 1);
- nell'ultimo caso non interviene alcun *autoboxing* e quel che accade è che l'invocazione viene indirizzata alla segnatura di arietà 1 che riceve un solo oggetto di tipo `int[]` e quindi costruisce una lista di tipo `List<int[]>` un solo elemento… dato da `arrayDiInt`!

Si può facilmente verificare che tale è il caso con

```
List<int[]> listaDiArrayDiInt = Arrays.asList(arrayDiInt);
int[] unicoArray = listaDiArrayDiInt.get(0);
unicoArray[1]
```

```
5
```

Da collezioni ad array

Nella direzione opposta, osserviamo che ciascun sottotipo di `Collection` ha un metodo (ereditato da) `toArray` che consente di ottenere un array di riferimenti agli elementi che contiene; la segnatura del metodo prevede che venga passato come argomento un array (anche vuoto) del tipo dell'array che si intende ottenere (questo è dovuto ad alcune particolarità del modo in cui interagiscono array e metodi generici). L'uso di tale metodo è elementare

```
List<Integer> interi = List.of(1, 2, 3);
Integer[] comeArray = interi.toArray(new Integer[0]);
Arrays.toString(comeArray)
```

```
[1, 2, 3]
```

Attenzione perché omettendo l'argomento sarà selezionato il metodo sovraccaricato `toArray` che restituisce un `Object[]` e non è possibile effettuare alcun cast diretto che lo renda un array di elementi di tipo diverso, come mostra l'esempio seguente

```
try {
  Integer[] comeArray = (Integer[])interi.toArray();
} catch (ClassCastException e) {
  System.err.println(e);
}
```

```
java.lang.ClassCastException: class [Ljava.lang.Object; cannot be cast to class
[Ljava.lang.Integer; ([Ljava.lang.Object; and [Ljava.lang.Integer; are in module java.base
of loader 'bootstrap')
```

Certamente si può effettuare una sorta di cast col metodo `copyOf` come suggerito in precedenza, ma ovviamente è più efficiente ottenere direttamente l'array del tipo desiderato.

Le mappe

Le mappe (che non sono sottotipi di `Collection`) non hanno un metodo che consenta di ottenerne direttamente il contenuto sotto forma di array; ogni mappa però può restituire l'insieme delle sue `Map.Entry` (ossia delle coppie chiave e valore), da cui può essere quindi un array; ad esempio

```
Map.Entry[] entries = mappa.entrySet().toArray(new Map.Entry[0])
```

tale array però non ha traccia dei parametri con cui era istanziata la mappa generica (e non si può istanziare un array di tipo parametrico); per questa ragione accedere a chiavi e valori richiede dei cast espliciti (invece di godere delle usuali garanzie offerte dai generici)

```
String chiave = (String)(entries[0].getKey());
Integer valore = (Integer)(entries[0].getValue());
chiave + "; " + valore
```

```
due; 2
```

La classe `Collections`

Per finire, analogamente al caso di `Objects` e `Arrays`, nella classe **`Collections`** ci sono una messe di metodi statici di utilità che possono risultare molto comodi nella prova pratica.

Collezioni vuote, riempimento e rimpiazzamento

In ossequio dell'Item 54 del Capitolo 8 del libro di testo “Effective Java”, è consigliabile restituire collezioni vuote, piuttosto che `null`. Per questa ragione `Collections` mette a disposizione una serie di metodi statici di nome `emptyT` dove `T` è il tipo di una collezione (ma anche un iteratore vuoto, ad esempio).

> **Suggerimento**
>
> Non sottovalutate la semplificazione consentita dall'accorgimento di usare collezioni vuote (se logicamente ammissibili per le specifiche) al posto di `null`. Se il metodo `aCollection` di una classe adottasse tale convenzione, ai suoi utilizzatori sarebbe consentito di scrivere, ad esempio
>
> ```
> for (AType e : aCollection()) doSomething(e);
> ```
>
> invece del più verboso
>
> ```
> Collection<AType> c = aCollection();
> if (c != null) for (AType e : c) doSomething(e);
> ```
>
> o di commettere un errore grave nel caso si omettesse, non avendo adottato la convenzione, il controllo di nullità

I metodi statici **`fill`** e **`replaceAll`** consentono, rispettivamente, di riempire una lista con un dato elemento, o rimpiazzare tutte le occorrenze di un elemento con un altro; il secondo potrebbe essere usato, ad esempio, per rimpiazzare i valori `null` con un “default”

```
List<String> paroleENull = Arrays.asList("uno", null, "due", null, null, "tre");
List<String> parole = new ArrayList<>(paroleENull);
Collections.replaceAll(parole, null, "");
parole
```

```
[uno, , due, , , tre]
```

Ordinare, cercare e contare nelle liste

Le osservazioni della omologa sezione per gli array si applicano in modo del tutto analogo per le liste; il metodo **`sort`** è però in questo caso di istanza e ha un'unica versione che ha un comparatore per argomento (che si intende quello dell'ordine naturale degli elementi della lista se `null`); inutile osservare che siccome l'ordinamento avviene anche in questo caso in loco, è necessario che la lista sia modificabile.

Qualche esempio concreto può aiutare a comprenderne l’uso: ordine naturale (specificando `null`)

```
listaModificabile.sort(null);
listaModificabile
```

```
[due, quattro, tre, uno]
```

stessa cosa con il metodo statico di `Comparator`

```
listaModificabile.sort(Comparator.naturalOrder());
listaModificabile
```

```
[due, quattro, tre, uno]
```

mentre per l’inverso dell’ordine naturale

```
listaModificabile.sort(Comparator.reverseOrder());
listaModificabile
```

```
[uno, tre, quattro, due]
```

In alcuni casi (ad esempio se è noto che non si riutilizzerà più un certo ordine) può essere molto comodo usare una classe anonima

```
listaModificabile.sort(new Comparator<>() {
  @Override
  public int compare(String o1, String o2) {
    if (o1.isEmpty()) return -1;
    if (o2.isEmpty()) return 1;
    return Character.compare(o1.charAt(o1.length() - 1), o2.charAt(o2.length() - 1));
  }
});
listaModificabile
```

```
[tre, due, uno, quattro]
```

in questo caso, le stringhe sono ordinate in base all’ordine alfabetico del loro ultimo carattere (se non vuote).

Come nel caso degli array, la ricerca di un elemento in una lista ordinata può essere effettuata con il metodo **`binarySearch`** se i suoi elementi sono comparabili, oppure specificando un comparatore con la versione del metodo che lo riceve come secondo argomento.

Vale la pena di ricordare che se lista non è ordinata può essere comunque effettuata una ricerca di un elemento (in tempo lineare) tramite il suo metodo d’istanza **`indexOf`** (o **`lastIndexOf`**).

Per cercare una *sottolista*, è invece possibile usare il metodo statico **`indexOfSubList`** (o lastIndexOfSubList) di `Collections`; ad esempio

```
listaModificabile.sort(null);
int idx = Collections.indexOfSubList(listaModificabile, List.of("tre", "uno"));
lista + "; " + idx
```

```
[uno, due, due]; 2
```

Nel caso delle liste, se non si è interessati all’ordine di tutti gli elementi, ma solo ai *valori estremi*, si possono utilizzare i metodi **`min`** e **`max`** che sono basati sull’ordine naturale, oppure le loro versioni che consentono di indicare un comparatore come secondo argomento.

Per finire, se si vuole contare il numero di occorrenze di un certo valore, si può adoperare il metodo **`frequency`**; ad esempio

```
int num =  Collections.frequency(paroleENull, null);
paroleENull + "; " + num
```

```
[uno, null, due, null, null, tre]; 3
```

riporta il numero di `null` nella collezione.

Mescolare e ruotare

Ci sono diverse operazioni comuni che agiscono sulle *posizioni* degli elementi di una collezione senza però modificarne gli elementi stessi:

- lo *scambio* di due elementi, ottenuto tramite il metodo **`swap`**;
- il *rovesciamento* della lista, ottenuto tramite il metodo **`reverse`**;
- la *rotazione* della lista, ottenuta tramite il metodo **`rotate`**, che agisce come se la lista fosse circolare e la testa fosse spostata di una data `distanza` (altrimenti detto l'elemento di posto *i* nella lista ruotata corrisponde a quello di posto *i* - `distanza` nella lista originale);
- il *mescolamento casuale* degli elementi, ottenuto tramite il metodo **`shuffle`**.

Riguardo all'ultimo metodo, è possibile specificare esplicitamente il generatore di *numeri pseudocasuali* usando la versione sovraccaricata **`shuffle`** che accetta una istanza della classe **`Random`** come argomento; questo è particolarmente importante per rendere *riproducibile* il comportamento

> **Suggerimento**
>
> Quando utilizzate un oggetto di tipo **`Random`** per introdurre della casualità nel comportamento del vostro codice può diventare molto arduo individuarne gli errori perché ogni esecuzione si comporta potenzialmente in modo diverso dalle precedenti.
>
> Fortunatamente, la classe `Random` implementa un *generatore lineare congruenziale* per cui specificando esplicitamente il *seme* usando il costruttore **`Random`** si otterrà sempre la stessa sequenza di esecuzione.
>
> Può essere quindi una buona idea istanziare il generatore tramite una funzione del genere
>
> ```
> static void Random reproducibleRng(long seed) {
>   if (seed == 0) {
>     seed = System.currentTimeMillis();
>     System.err.println("reproducibleRng: seed = " + seed);
>   }
>   return new Random(seed);
> }
> ```
>
> in questo modo, se la funzione è invocata con un valore nullo del seme essa ne sceglierà uno (ogni volta diverso, grazie alla chiamata di **`currentTimeMillis`**) e ne emetterà il valore sul flusso d'errore.