CORSO JAVA AESYS
Parte Prima

2022

www.aesystech.it

Collections Framework e Stream API parte prima

- Espressioni Lambda
- Framework Collections
- L’interfaccia Collection
- Iterare sulle collezioni
- Interfaccia List

Espressioni Lambda

Espressione Lambda deriva dal «Lambda Calcolo» da cui ebbe inizio in qualche modo la storia della programmazione. Un'espressione lambda è detta anche funzione anonima (in inglese anonymous function). Questo perché in effetti si tratta proprio di una funzione, quindi non un metodo appartenente ad una classe e chiamato tramite un oggetto, ma una funzione senza nome definita al volo in maniera simile a come abbiamo fatto con le classi anonime.
La sintassi di un'espressione lambda è forse la più complicata tra le sintassi di elementi definite da Java. Infatti è possibile sintetizzare questa sintassi omettendo tutto quanto il compilatore può dedurre da solo. In generale la sintassi è la seguente:

```
([lista di parametri]) -> {blocco di codice}
```

Il primo elemento è una lista di parametri. Nell'esempio precedente la lista di parametri era vuota, ed è stata esplicitata con un blocco di parentesi tonde vuote. Poi segue una "freccia", costituita dal segno di sottrazione "–", concatenato con il simbolo maggiore ">".

Comprendere le espressioni lambda

Quando si ha a che fare con le espressioni lambda, all'inizio si può rimanere spiazzati. Come abbiamo visto, la sintassi ci permette di evitare di scrivere parentesi tonde, tipi che si possono dedurre, comandi di tipo return e parentesi graffe. Questo perché questa nuova funzionalità vuole essere soprattutto un modo per abbassare la verbosità del linguaggio. Se da un lato scriviamo meno codice, dall'altro potremmo perdere in leggibilità. Il segreto è quello di non dimenticare mai l'equivalenza tra classe anonima ed espressione lambda. Una volta fatto proprio il concetto che un'espressione lambda possa essere referenziata con un reference di un'interfaccia funzionale, e che la sua definizione non è altro che l'implementazione dell'unico metodo astratto (SAM) dell'interfaccia funzionale, siamo già ad un ottimo punto per una buona comprensione di questo argomento. Ripetiamo, l'unica cosa che può fare un'espressione lambda è **sostituire un'interfaccia funzionale.**

Dovremmo usare le espressioni lambda quando il nostro obiettivo è quello di passare in maniera dinamica un certo algoritmo ad un altro metodo. Questo serve per eseguire l'algoritmo in un contesto definito dal metodo a cui stiamo passando l'algoritmo. Passiamo del codice ad un metodo, per dare a quest'ultimo la responsabilità di chiamarlo al momento giusto.

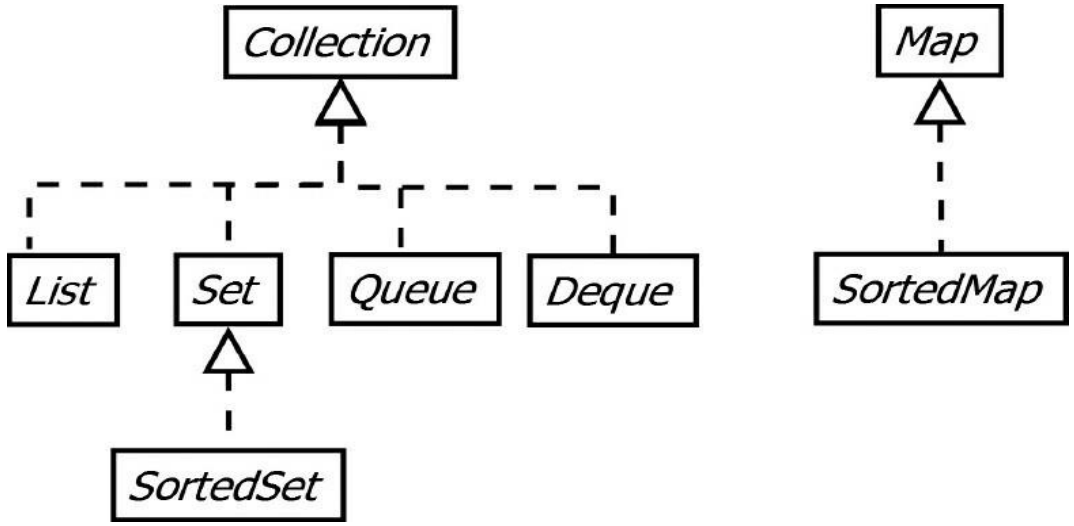
Potremmo anche passare del codice ad un metodo per farlo eseguire in base ad un altro evento come il lancio di un'eccezione. Oppure potremmo anche fare in modo che il nostro codice sia eseguito mentre si itera su ogni elemento di un array. In generale passare una espressione lambda, significherà far decidere al metodo a cui abbiamo passato il codice, il momento in cui eseguirlo.

Reference a metodo

Un reference a metodo può essere passato come parametro ad un metodo, esattamente come una espressione lambda. È molto utile utilizzare un reference a metodo in luogo di una espressione lambda quando questo deve essere riutilizzato, per evitare di scrivere più volte le stesse righe di codice. Inoltre se ci sono troppe righe di codice da definire, è preferibile utilizzare un reference a metodo piuttosto che un'espressione lambda. Esistono quattro tipi di sintassi diverse per utilizzare i reference a metodo. La sintassi del reference a un metodo statico permette di passare un metodo definito statico come parametro ad un metodo. In questo caso si utilizza il nome del tipo per referenziare il metodo, seguito come sempre dalla coppia
:: e dal nome del metodo statico: **nomeTipo::nomeMetodoStatico**. Stesso discorso per un reference a un metodo d'istanza, dove però è necessario istanziare un oggetto ed utilizzare il suo reference prima della coppia ::, seguito dal nome del metodo d'istanza: **nomeOggetto::nomeMetodoDIstanza**. Si può utilizzare la sintassi di un reference a un metodo d'istanza di un certo tipo, quando il compilatore può dedurre senza l'istanza di un particolare oggetto, quali sono gli input e gli output del metodo. In questo caso si utilizza il nome del tipo, la coppia :: e il nome di un metodo non statico: nomeTipo::nomeMetodoDIstanza. Infine è possibile anche utilizzare la sintassi per il reference ad un costruttore, che consente di passare una nuova istanza come parametro ad un metodo. La sintassi utilizza il nome del tipo, la coppia :: e infine la parola chiave new: **nomeTipo::new.**

Introduzione al Framework Collections 1/2

Il framework noto come Collections è costituito da una serie di classi, interfacce e algoritmi per gestire collezioni. Una collezione è definibile come un oggetto che raggruppa più elementi in una singola entità. I vantaggi di avere a disposizione questo framework per la programmazione sono tanti: possibilità di scrivere meno codice, incremento delle prestazioni, interoperabilità tra classi non relazionate tra loro, riusabilità, algoritmi complessi già implementati a disposizione, maggiore qualità e semplicità del codice, estensibilità per creare nuove librerie.
Il framework è basato su otto interfacce principali che vengono poi estese da tante altre classi astratte e concrete. In figura viene illustrata, tramite un diagramma delle classi UML semplificato, la gerarchia di queste interfacce che è costituita da due alberi diversi.



Introduzione al Framework Collections 2/2

Ad un primo livello si trovano le interfacce Collection (la più importante) e
Map. Collection è estesa dalle interfacce List, Queue, Deque e Set. Quest'ultima a sua volta è estesa da SortedSet. Map è invece estesa da
SortedMap. Ognuna di queste interfacce possiede proprietà che le sottoclassi ereditano. Per esempio, una classe che implementa Set (in italiano "insieme") erediterà il fatto di essere un tipo di collezione non ordinata che non accetta elementi duplicati. Cerchiamo di fare chiarezza con una breve panoramica su queste interfacce e le loro proprietà.
L'interfaccia Collection è la radice principale di tutta la gerarchia del framework. Essa astrae il concetto di insiemi di oggetti, detti elementi. Esistono implementazioni che ammettono elementi duplicati ed altre che non lo permettono, collezioni ordinate e non ordinate. La libreria non mette a disposizione alcuna implementazione diretta di Collection, ma solo delle sue dirette sottointerfacce come Set e List. L'interfaccia Collection viene definita come il minimo comune denominatore che tutte le collection devono implementare. Un Set è un tipo di collection che, astraendo il concetto di insieme matematico, non ammette elementi duplicati. Una List è una collezione ordinata (a volte viene detta anche sequence). In una lista viene sempre associato un indice ad ogni elemento, che equivale alla posizione dell'elemento stesso all'interno della lista. Una lista ammette elementi duplicati (distinguibili fra loro per la posizione).

L’interfaccia Collection

È la superinterfaccia per definizione del framework. Possiamo usare un reference di Collection per puntare a tutte le implementazioni che appartengono allo stesso albero gerarchico per il polimorfismo per metodi. Per esempio possiamo scrivere:

```
Collection<String> collection = new ArrayList<>();
```

Questa pratica che sfrutta il polimorfismo per dati è sempre consigliabile, e ancora di più per le collections. Per esempio se avessimo scritto tante altre righe di un programma che utilizzano la variabile collection, e avessimo necessità di voler cambiare il tipo dell’oggetto per esempio con una LinkedList, l’unica riga da cambiare sarebbe proprio quella appena scritta. Tutte le altre parti di codice che usano collection continueranno a funzionare.
Essendo ArrayList e LinkedList implementazioni di List (che a sua volta estende Collection), a seconda dei casi, è possibile che sia più utile utilizzare il polimorfismo per dati usando tale interfaccia piuttosto che Collection. Per esempio se vogliamo sfruttare il metodo get() definito nell’interfaccia List, che prende in input un intero rappresentante l’indice dell’elemento della lista che si vuole recuperare, allora è preferibile usare un reference di List piuttosto che di Collection:

```
List<String> list = new ArrayList<>(); list.add("Primo elemento");
list.add("Secondo elemento");
String primoElemento = list.get(0);
```

Metodi di Collection

L'interfaccia Collection definisce metodi di base come:

- int size() ritorna il numero degli elementi contenuti nella collezione.
- boolean add(E element) aggiunge un elemento (del tipo parametro) alla collezione, e restituisce true o false a seconda del fatto che l'operazione sia andata a buon fine o meno.
- boolean remove(Object element) rimuove un elemento dalla collezione, e restituisce true o false a seconda del fatto che l'operazione sia andata a buon fine o meno.
- boolean isEmpty() restituisce true o false a seconda del fatto che la collection contenga elementi o meno.
- void clear() elimina tutti gli elementi dalla collezione.
- boolean contains(Object element) restituisce true o false a seconda del fatto che l'elemento specificato in input sia contenuto nella collection o meno. Per verificare l'uguaglianza viene utilizzato il metodo equals().
- Iterator<E> iterator() restituisce un'implementazione di un'interfaccia Iterator che serve per iterare sugli elementi della collezione.
- Collection definisce metodi per interagire con altre collezioni (il che significa con implementazioni di List, Set e così via.
- Collection definisce anche metodi per interagire con array

Iterare sulle collezioni

Esistono tre modi per iterare sulle collezioni:

1. usando cicli foreach;
2. usando iteratori;
3. usando il metodo di default forEach() definito nell'interfaccia Collection e sfruttando le cosiddette **operazioni di aggregazione** (in inglese **aggregate operations)** della nuova libreria denominata Stream API.

Con l'avvento di Java 8, è stato introdotto il metodo forEach() come metodo di default nell'interfaccia Iterable (da non confondere con il costrutto che di solito chiamiamo foreach). Il metodo forEach() prende in input un oggetto di tipo Consumer (cfr. paragrafo 16.3.2) che ricordiamo, ci permette di eseguire operazioni su oggetti (di solito per modificarne lo stato interno). Questo significa che qualsiasi collezione potrà iterare direttamente sui suoi elementi semplicemente chiamando questo metodo, e passandogli in input un'espressione lambda per fare operazioni sugli stessi elementi. Per esempio, considerando l'oggetto smartphones dell'esempio precedente, con il seguente statement:

```
smartphones.forEach(s->System.out.println(s));
```

Interfaccia List

L'interfaccia List rappresenta una collezione ordinata e indicizzata di elementi in cui sono ammessi duplicati. Estende Collection, che a sua volta estende Iterable, e naturalmente eredita tutti i metodi pubblici da entrambe. In generale una lista definisce i metodi add() e addAll() (che ricordiamo prende in input una collection), in modo tale che aggiungano in coda gli elementi in input. La sequenza di come gli elementi di una lista vengono aggiunti quindi, definisce l'ordine della lista. Uno dei metodi più utilizzati in assoluto è il metodo get() che prende in input un indice intero e restituisce l'oggetto corrispondente a quell'indice. Si usa per esempio quando si cicla su una lista con un ciclo for sfruttando l'indice della collection:

```
List<Integer> list = new ArrayList<>(3); list.add(25);
list.add(7); list.add(74); int size = list.size();
for (int i = 0; i < size; i++) {
   System.out.println(list.get(i)); }
```

Le principali implementazioni di List sono ArrayList, e LinkedList. ArrayList è in assoluto la collection più utilizzata, ed è stata creata proprio per rappresentare una evoluzione ridimensionabile di un array.ArrayList ha prestazioni nettamente superiori anche a LinkedList, che conviene utilizzare solo per gestire collezioni di tipo coda. Si tratta di una cosiddetta lista concatenata, nella quale ogni elemento mantiene un reference verso l'elemento successivo e l'elemento precedente. Mette a disposizione tra l'altro metodi come addFirst(), getFirst(), removeFirst(), addLast(), getLast() e removeLast(). È quindi opportuno scegliere l'utilizzo di una LinkedList in luogo di ArrayList solo quando si devono aggiungere spesso elementi all'inizio della lista.