Programmazione Java

Seconda Parte

Alfonso Domenici

Argomenti

Interfacce

Tipi generici

Iteratori

Metodi generici

Ordinamenti e interfacce relative

Interfacce funzionali e lambda expression

Collezioni: set, liste, mappe

Stream

Interfacce

Un'interfaccia stabilisce delle caratteristiche comportamentali mediante la dichiarazione di metodi astratti (come le classi astratte).

C'è un legame di implementazione tra classi e interfacce: una classe, se dichiara di implementare un'interfaccia, deve implementarne tutti i metodi astratti.

Un'interfaccia può definire il tipo di un riferimento, come una classe.

Un riferimento di tipo interfaccia punta ad un oggetto di una classe che implementa l'interfaccia: i metodi applicabili all'oggetto sono soltanto quelli dichiarati nell'interfaccia.

Oggetti di classi diverse, che però implementano la stessa interfaccia I, possono essere accessibili tramite lo stesso riferimento di tipo I.

Un'interfaccia semplice

```
public interface MediaPlayer {
    void play(int volume);
}
public class CDPlayer implements MediaPlayer{
    ...deve riscrivere tutti i metodi previsti dall’interfaccia
}
```

Riepilogo

Le interfacce non possono essere istanziate ; possono essere implementate da classi o estese da altre interfacce.

Una classe può implementare più di una interfaccia.

Un'interfaccia può essere il tipo di un riferimento; allora quel riferimento può puntare soltanto ad un oggetto la cui classe implementi l'interfaccia.

Le interfacce possono contenere anche costanti, metodi default (con body), metodi statici (con body) e tipi annidati.

I metodi sono implicitamente pubblici.

Le costanti sono implicitamente public, static e final.

Metodi default

Se ad un'interfaccia già in uso si aggiunge un metodo default, non occorre aggiornare le classi che la implementano; è comunque possibile farlo per ridefinire il metodo default.

Esempio

```
public interface MediaPlayer {
    void play(int volume);
    int maxVolume = 100;
    default boolean checkVolume(int volume) {
        return volume < maxVolume;
    }
}
```

Metodi statici

```
public interface CommonInfo {
	String getNome();
	long getValore();
	default String getInfo() {
		return getNome() +": " + getValore() + " ";
	}
	static void stampaListaCommonInfo (String nomeLista, CommonInfo[] lista){
		System.out.println(nomeLista);
		for ( CommonInfo i:lista) System.out.print(i.getInfo());
	}
}
```

Tipi generici

Fondamentali per costruire contenitori di oggetti di tipi non noti a priori .

La struttura e la gestione di questi contenitori è indipendente dai tipi degli oggetti contenuti.

Stanno alla base delle collezioni di libreria.

Esempi: contenitore semplice, stack

Visita degli elementi di un contenitore mediante un iteratore

Interfacce Iterable e Iterator

Classi annidate

Digressione: classi annidate statiche

Metodi generici

Restrizioni applicate ai tipi

Il costrutto diamond <>

Tipi generici

Un tipo generico è una classe o interfaccia che ha dei parametri (uno o più) e questi parametri sono tipi non noti a priori ; pertanto sono indicati con lettere maiuscole (ad es. E, T).

Una classe generica definisce una struttura per organizzare oggetti omogenei indipendentemente dal loro tipo.

Esempio: public class Box <T> { ... }

Per usare questa classe occorre specializzarla con un tipo non generico

Es. Box <String> box;

Box <String> non è più una classe generica, ma una classe specializzata che si può usare per definire oggetti come le classi normali.

Analogamente si può definire un'interfaccia generica, mentre i tipi enumerativi non possono essere generici.

Vedere tipo Optional in java.util

Esercizio

Si costruisca uno stack generico, Stack <T> o Pila, le cui proprietà sono definite nell'interfaccia generica seguente:

Nota:

Inoltre il costruttore dello stack permette di stabilire la capacità dello stack; il metodo put dà false se lo stack è pieno e get dà null se è vuoto.

Esempi: Iteratori

Se una classe definisce una struttura di elementi, spesso deve consentire agli utenti l'accesso agli elementi (visita) in modo ordinato, uno alla volta.

Per gestire la visita la classe fornisce un iteratore , che è un oggetto capace di mantenere lo stato della visita. Definito con una interfaccia Generica Iterator

Interface Iterator<E>

    boolean hasNext();

    E next();

L'iteratore è fornito dalla classe in modo standard attraverso l'interfaccia generica .

Interface Iterable<T>

    Iterator<T> iterator()

Esercizio: Stack visitabile

Se lo stack è visitabile si possono leggere gli elementi in due modi: forma compatta o forma estesa.

Ciclo for(...) oppure tramite utilizzo dell’Iterator in modo esplicito..

Lo stack deve fornire un oggetto iteratore (che implementa l'interfaccia Iterator con i metodi hasNext e next). L'oggetto iteratore tiene lo stato dell'iterazione.

La sua classe non è di interesse per il chiamante, quindi si può definire come classe annidata nella classe dell'oggetto visitabile

Classi Annidate

All'interno di una classe annidata sono visibili le proprietà (anche private) della classe principale e la classe principale può vedere le proprietà anche private di una classe annidata.

Un oggetto di una classe annidata è quindi collegato implicitamente all'oggetto della classe principale che l'ha generato.

Classi annidate statiche

Una classe annidata può essere statica: in questo caso i suoi oggetti non sono legati a quelli della classe che la contiene e pertanto può essere istanziata direttamente dall'esterno, se è visibile.

Metodi generici

Un metodo generico è preceduto da uno o più parametri che sono tipi non noti a priori (indicati con lettere maiuscole).

Es.

public static <T> Stack<T> unioneStack (Stack<T> s1, Stack<T> s2);

Esercizio: realizzare tale metodo

Restrizioni applicabili ai parametri che sono tipi

Scrivere un metodo statico che sommi il contenuto di uno stack numerico. Bisogna restringere T ad un sottoinsieme di tipi numerici.

Due modi:

public static <T extends Number> double totale (Stack<T>> s)

public static double totale (Stack<? extends Number> s) {

La variante si può usare quando il parametro tipo compare soltanto nella restrizione.

<? extends Number> indica una classe che può essere Number o una sua sottoclasse (Upper Bounded Wildcard)

Restrizioni con wildcards

Restrizione verso l'alto dell'albero di inheritance: da X in giù

<? extends X> indica una classe che può essere X o una sua sottoclasse (Upper Bounded Wildcard)

Restrizione verso il basso: da X in su

<? super X> indica una classe che può essere X o una sua superclasse (Lower Bounded Wildcard)

X può essere un tipo generico oppure un tipo normale.

Il costrutto diamond <>

Stack<Integer> intStack1 = new Stack<>(3);

Il compilatore esegue una type inference e quindi assume che la dichiarazione sia

Stack<Integer> intStack1 = new Stack<Integer>(3);

Nota: in situazioni complesse , come quelle che si possono avere negli stream, il compilatore potrebbe non riconoscere i tipi sottintesi, quindi è meglio evitare l'uso del diamond.

Ordinamenti e interfacce relative

Significato

Criterio naturale: interface Comparable <T>

Altri criteri: interface Comparator <T>

Implementazione dei criteri

Classi anonime

Ordinamento di array: metodo Arrays.sort

Interfacce per ordinamenti

Spesso occorre ordinare una sequenza di oggetti. I criteri di ordinamento possono essere vari: ad es. una sequenza di stringhe può essere ordinata in senso alfabetico crescente o decrescente. Il criterio principale è detto naturale, gli altri sono aggiuntivi.

Il criterio naturale è stabilito dall'interfaccia seguente (in java.lang)

Interface Comparable <T>

    int compareTo (T o)

Tutte le classi di libreria implementano questa interfaccia e quindi stabiliscono i criteri naturali.

Il metodo sort di Arrays ordina con il criterio "naturale" che è associato alla classe degli elementi da ordinare (alfabetico per String).

Se si vuole ordinare con un altro criterio?

Si può usare un oggetto comparatore, cioè un oggetto che implementa l'interfaccia seguente (in java.util):

interface Comparator<T>

    int compare(T o1, T o2)

Esiste un'altra versione di sort

static <T> void sort(T[] a, Comparator<? super T> c) che ha come secondo argomento un comparatore.

Si può definire un comparatore in 3 modi:

con una classe normale,

con una classe anonima,

con una lambda expression.

Esempio con classe anonima

Il costrutto

```
new Comparator<String>() {
      public int compare(String s1, String s2)
      {return s2.compareTo(s1);}
};
```

produce una classe anonima (per l'utente) che implementa l'interfaccia Comparator<String> e chiama il costruttore di default di questa classe.

L'oggetto fornito è visto attraverso l'interfaccia.

Interfacce funzionali e lambda expression

Caratteristiche delle interfacce funzionali

Caratteristiche delle espressioni lambda

Interfacce funzionali di uso comune in java.util.function

Method reference

Metodi statici di BinaryOperator <T>

Metodi statici di Comparator <T>

Esempi

Interfacce funzionali e lambda expression

Soltanto se l'interfaccia è funzionale, si può implementare con una lambda expression.

Un'interfaccia funzionale contiene un solo metodo abstract.

Sono esclusi quelli che ridichiarano i metodi di Object.

L'interfaccia Comparator<T> è funzionale perché contiene un solo metodo abstract

```
int compare(T o1, T o2)
```

Uso di lambda expression

Si può scrivere quindi

Comparator<String> comp = (s1, s2) -> s2.compareTo(s1);

Dato il contesto (la parte sinistra), si capisce che si tratta dell'implementazione del metodo compare.

La forma di una lambda è: parametri -> body.

Nel caso considerato i parametri sono due stringhe e il body è un'espressione basata sui parametri che dà un risultato del tipo atteso (int).

Una lambda denota quindi un oggetto (di una classe anonima) che implementa un'interfaccia funzionale.

Interfacce funzionali di usi comune: package java.util.function

Esempi

```
Consumer<Object> cons1 = x -> System.out.println(x);
cons1. accept ("alfa"); // alfa

Function <String, String> f1 = x -> x.toUpperCase();
String r = f1. apply ("alfa");
System.out.println(r); //ALFA
```

Method reference

Implementa una function in modo più semplice:

nome classe :: nome metodo .

x -> x.toUpperCase() diventa String::toUpperCase

quindi

Function <String, String> f1 = String::toUpperCase;

Altri esempi…

Function

Se il risultato di una function è usato come chiave per un ordinamento,si chiama sort key e la funzione si chiama key extractor .

Interfaccia Comparator <T>

Contiene dei metodi statici che generano dei comparatori e semplificano quindi la scrittura di regole di confronto complesse; la performance risulta però inferiore.

metodi statici principali

comparing

comparingInt, comparingLong, comparingDouble

thenComparing

thenComparingInt, thenComparingLong, thenComparingDouble

naturalOrder

reverseOrder

reversed

Esempio

```
import static java.util.Comparator.*;


Comparator<Point> comp1 =
    comparing(Point::getX, reverseOrder())
    .thenComparing(Point::getY, reverseOrder());
```

Comparing può avere uno o due parametri; in questo caso 2.

Il primo parametro è un key extractor che dà la sort key, il secondo è un comparatore che confronta le sort keys

Si usa import static per poter usare le proprietà statiche della classe direttamente (senza farle precedere dal nome della classe).

Collezioni

Significato

Insiemi, liste e mappe

Interfacce e classi

Classe Collections

Ordinamenti

Collezioni

Sono strutture generiche: insiemi (set), liste e mappe.

Le implementazioni sono date da classi generiche e le funzionalità sono indicate in interfacce generiche.

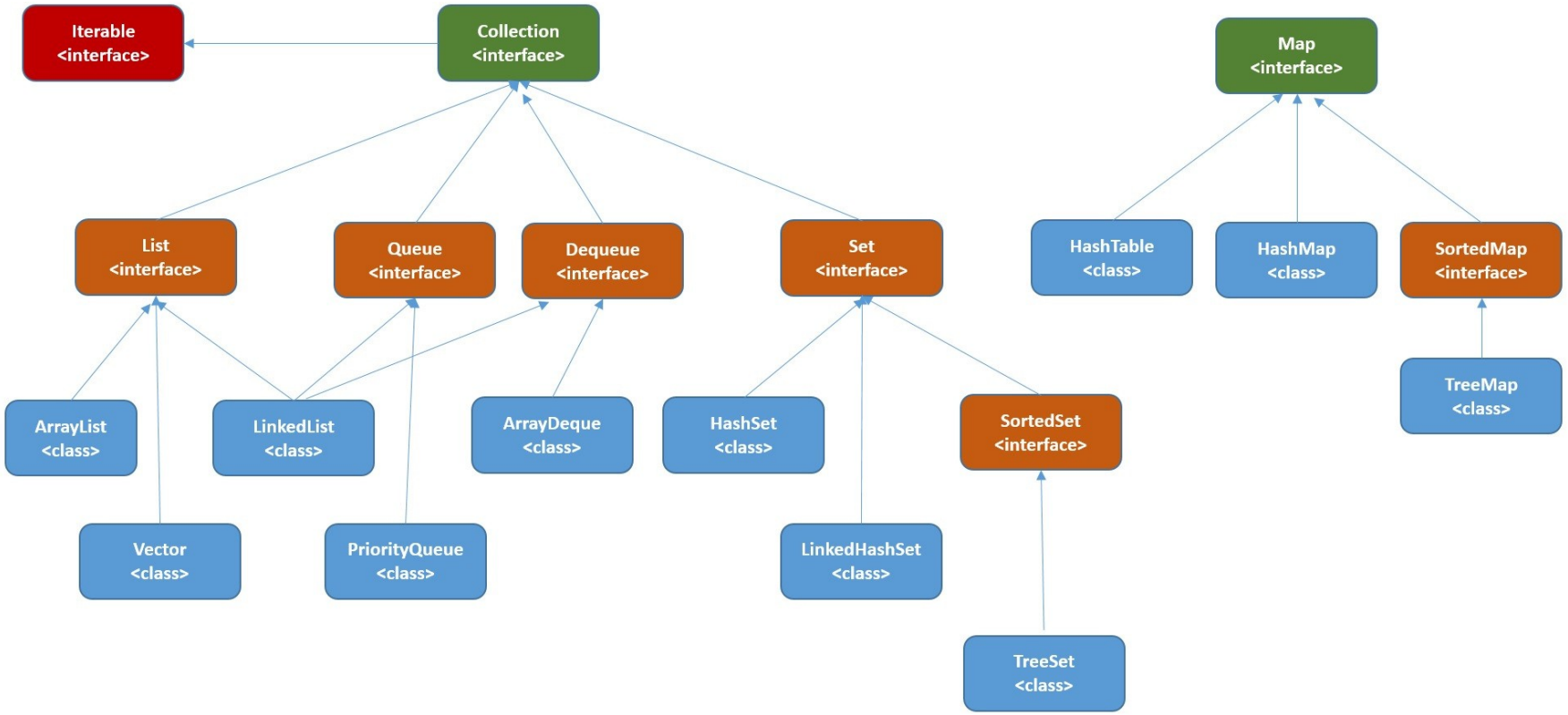
Algoritmi di manipolazione come il sorting sono forniti dalla classe Collections.

Set (insieme): contiene elementi distinti

Lista: sequenza di elementi con posizione, da 0 in avanti

Mappa: coppie chiave – valore

Collection Framework Hierarchy



Collection<E>

metodi di base

int size (), boolean isEmpty(), boolean contains (Object element),

boolean add (E element), boolean remove(Object element),

Iterator<E> iterator().

metodi che operano su collezioni

boolean containsAll(Collection<?> c), boolean addAll(Collection<? extends E> c),

boolean removeAll(Collection<?> c), boolean retainAll(Collection<?> c),

void clear(), boolean removeIf(Predicate<? super E> filter).

metodi che copiano la collezione in un array

Object[] toArray(), <T> T[] toArray(T[] a.

metodi per l'uso di stream

Stream<E> stream () and Stream<E> parallelStream().

Note

boolean add(E element) dà true se ha aggiunto l'elemento alla collezione, altrimenti dà false.

OPERAZIONI INSIEMISTICHE

addAll aggiunge la seconda collezione alla prima (unione)

removeAll toglie dalla prima collezione gli elementi che si trovano nella seconda (differenza)

retainAll tiene nella prima collezione soltanto gli elementi che si trovano anche nella seconda (intersezione)

containsAll dà true se la prima collezione contiene tutti gli elementi della seconda (inclusione)

Set

L'interfaccia Set<E> contiene gli stessi metodi di Collection<E>.

Ci sono varie implementazioni tra cui HashSet, TreeSet e LinkedHashSet.

Un set contiene elementi distinti. Un elemento duplicato non è inserito e il metodo add dà false come risultato.

Negli HashSet gli elementi non sono ordinati e il controllo della duplicazione si basa sulla ridefinizione dei metodi hashCode e equals nelle classi degli oggetti contenuti.

Nei TreeSet si possono avere due tipi di ordinamento, naturale oppure esterno (mediante comparatore).

L'ordinamento naturale è basato sull'interfaccia Comparable .

Un LinkedHashSet mantiene l'ordine di inserimento.

SortedSet<E>

public interface SortedSet<E> extends Set<E>

Un TreeSet implementa l'interfaccia SortedSet che definisce questi metodi:

E first ()

E last ()

SortedSet<E> headSet(E toElement) // subset fino a toElement escluso

SortedSet<E> tailSet(E fromElement) // subset da fromElement incluso

SortedSet<E> subSet(E fromElement, E toElement) // subset da fromElement incluso a toElement escluso

Esempi sui set

Requisiti

Dato un elenco di parole, si vogliono ottenere:

- le parole distinte ordinate,
- le parole distinte che compaiono con ripetizioni nell'elenco,
- le parole distinte che compaiono senza ripetizioni nell'elenco,
- la prima parola dell'elenco ordinato.

Liste

Una lista è una sequenza di elementi anche duplicati. Oltre ai metodi di Collection, List<E> fornisce metodi di

- accesso in base alla posizione: get, set, add;
- ricerca: indexOf, lastIndexOf;
- iterazione: listIterator (avanti o indietro);
- gestione di sottoliste: subList.

La classe Collections fornisce molti algoritmi per il trattamento di liste, tra i quali: sort, shuffle, reverse, swap, replaceAll, fill, copy, binarySearch.

Ci sono varie implementazioni tra cui ArrayList, LinkedList e PriorityQueue.

Alcuni metodi di List<E>

boolean add(E e) aggiunge come ultimo elemento

void add (int index, E element) inserisce l'elemento nella posizione indicata da index e trasla i successivi

E get (int index) legge l'elemento dato l'indice

E remove (int index) toglie l'elemento dato l'indice

E set (int index, E element) sostituisce l'elemento nella posizione indicata

Nota: se ad un indice non corrisponde nessun elemento, il metodo solleva l'eccezione runtime java.lang.IndexOutOfBoundsException.

Alcuni metodi di List<E>

int indexOf (Object o) dà l'indice della prima occorrenza dell'elemento, oppure -1 se manca

ListIterator<E> listIterator ()

ListIterator<E> listIterator (int index)

List<E> subList (int fromIndex, int toIndex) dà una vista della porzione di lista da fromIndex incluso a toIndex escluso.

Vedere interfaccia ListIterator..

Code

Ordinano gli elementi in vari modi.

ArrayBlockingQueue (in java.util.concurrent) ordina in modo FIFO (head è il primo elemento in coda e tail è l'ultimo); è una coda limitata (bounded buffer) basata su un array ed è usata nella programmazione concorrente.

PriorityQueue dispone gli elementi secondo l'ordinamento naturale oppure mediante un comparatore.

Queste classi implementano l'interfaccia Queue che definisce metodi per aggiungere un elemento in fondo alla coda, e per leggere o rimuovere il primo elemento.

L'interfaccia Deque tratta inserimento, lettura e rimozione ad entrambi gli estremi.

LinkedList e ArrayDeque le implementano entrambe e possono quindi operare come code (FIFO) o stack (LIFO).

Queue<E>

Questa interfaccia definisce 3 operazioni in due versioni.

aggiunta di un elemento:	add	offer
lettura del primo elemento:	element	peek
rimozione del primo elemento:	remove	poll

I metodi della prima versione (add, element, remove) possono sollevare eccezioni; quelli della seconda (offer, peek, poll) danno un risultato particolare in caso di errore.

Se la coda è vuota, remove ed element sollevano NoSuchElementException, mentre poll e peek danno null.

Se la coda è piena, add solleva IllegalStateException, mentre offer dà false.

Le classi LinkedList e PriorityQueue implementano questa interfaccia. PriorityQueue non accetta elementi nulli.

Deque<E>

Questa interfaccia rappresenta liste che consentono aggiunte, letture e rimozioni in testa e in coda (deque = double-ended queue). Deque si pronuncia come deck.

Ci sono quindi 6 metodi in due versioni :

1. addFirst, addLast, removeFirst, removeLast, getFirst, getLast.

2. offerFirst, offerLast, pollFirst, pollLast, peekFirst, peekLast.

L'interfaccia è implementata da LinkedList a da altre classi tra cui ArrayDeque.

Mappe

Le mappe sono set di coppie chiave-valore (entrambi oggetti).

Sono molto usate per gestire dati in memoria.

Spesso la chiave è un attributo (univoco) del valore; per comodità tale attributo è indicato con id.

Per controllare se una mappa contiene già un valore, si verifica se tra le chiavi compare il suo id.

Per aggiungere una coppia chiave-valore si usa il metodo put; per ottenere un valore data la chiave si usa il metodo get.

Il metodo values dà la collezione dei valori.

L'interfaccia delle mappe è Map<K,V>, dove K è il tipo delle chiavi e V il tipo dei valori.

L'interfaccia delle coppie chiave-valore è Map.Entry<K,V>.

Mappe

Una mappa definisce una funzione cioè una corrispondenza chiave- valore dove chiavi e valori sono oggetti e le chiavi non sono duplicate.

Quindi le chiavi di una mappa formano un set e i valori una collezione (in senso generico).

Offre 3 viste: il set delle chiavi, la collezione dei valori e il set delle coppie chiave-valore (il cui tipo è Map.Entry<K,V>).

Implementa metodi di base (put , get , remove, containsKey, containsValue, size, empty), metodi di massa (putAll, clear), metodi che

danno l'accesso alle 3 viste (keySet, entrySet, values).

Ci sono varie implementazioni tra cui HashMap, TreeMap, LinkedHashMap.

Con una treeMap si ha l'ordinamento delle chiavi in quanto inserite in un TreeSet.

Alcuni metodi di Map<K,V>

boolean containsKey(Object key)

boolean containsValue(Object value)

V get(Object key): dà il valore data la chiave oppure null se manca la chiave

V put(K key, V value): se manca la chiave aggiunge chiave e valore e dà null; altrimenti sostituisce il valore e dà quello precedente

V remove(Object key): se manca la chiave dà null; altrimenti rimuove chiave e valore e dà il valore

Alcuni metodi di Map<K,V>

Set<Map.Entry<K,V>> entrySet(): dà il set delle coppie chiave-valore

Set<K> keySet(): dà il set delle chiavi

Collection<V> values (): dà la collezione dei valori

default void forEach(BiConsumer<? super K,? super V> action): esegue un'azione per ciascuna coppia chiave-valore

void putAll(Map<? extends K,? extends V> m): aggiunge la mappa m alla mappa corrente

Interfaccia Map.Entry<K,V>

K getKey()

V getValue()

V setValue(V value)

Uso delle mappe nelle strutture dati

Gli oggetti dello stesso tipo sono solitamente identificati da una chiave univoca che è un attributo dell'oggetto. Per facilità di accesso gli oggetti sono collocati in mappe.

Sono frequenti due tipi di controllo: la presenza o l'assenza di un oggetto con una data chiave.

Il primo controllo permette di segnalare che si sta cercando di inserire un oggetto con una chiave già presente; il secondo che si sta cercando un oggetto inesistente.

L'esempio seguente mostra come effettuare questi controlli.

Esercizio..

Stream

Significato

Interfacce

Operazioni

Classe Optional

Collettori

Esempi con oggetti strutturati: lista di libri

Analisi degli ordini cliente con flatMap

Generazione degli ordini editore dagli ordini cliente

Uso di stream per fornire informazioni tramite un'interfaccia

Riepilogo

Approfondimenti

Stream

Uno stream è una sequenza di elementi omogenei: valori numerici (int, long, double) oppure oggetti.

La sorgente di uno stream può essere una collezione, un array oppure un file testuale.

Uno stream può essere sottoposto a varie operazioni definite nelle interfacce corrispondenti ai tipi di stream.

Le interfacce si trovano nel package java.util.stream e sono le seguenti:

IntStream, LongStream, DoubleStream, Stream<T>.

Operazioni di uno stream

Le operazioni si dividono in intermedie e finali.

Quelle finali danno come risultato un valore, un oggetto o una collezione oppure eseguono un'elaborazione.

Le operazioni intermedie possono essere concatenate con l'operatore "." in quanto danno come risultato lo stesso stream.