2018
Programmazione a oggetti
parte 2

Giorgio Bruno

Dip. Automatica e Informatica
Politecnico di Torino
email: giorgio.bruno@polito.it

Argomenti

Interfacce
Tipi generici
Iteratori
Metodi generici
Ordinamenti e interfacce relative
Interfacce funzionali e lambda expression
Collezioni: set, liste, mappe
Stream

Interfacce

Significato

Metodi astratti

Metodi default

Metodi statici: usati per fornire implementazioni di uso frequente

In seguito si studieranno le interfacce funzionali

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 Movable {
        void move(int x, int y);
}
```

L'interfaccia movable contiene soltanto il metodo astratto move (implicitamente public e abstract).

Se dichiariamo che

public class Point **implements** Movable { ...

public class Rectangle **implements** Movable { ...

ciò è vero in quanto esse implementano il metodo move.

Programma di test

```
public static void main(String[] args) {
        Point p = new Point(23, 94);  // il riferimento p ha per tipo una classe
        Particle pa = new Particle(10,20,100);
        Rectangle r = new Rectangle (p, 1000,2000);
        Movable m; // il riferimento m ha per tipo un'interfaccia
        m = p; m.move(1,2); //unico metodo utilizzabile
                                System.out.println(p);
        m = r; m.move(1,2); System.out.println(r);
        m = pa; m.move(1,2); System.out.println(pa);
        Movable[] a = {p, r, pa};
                System.out.println(Arrays.toString(a));
}
```

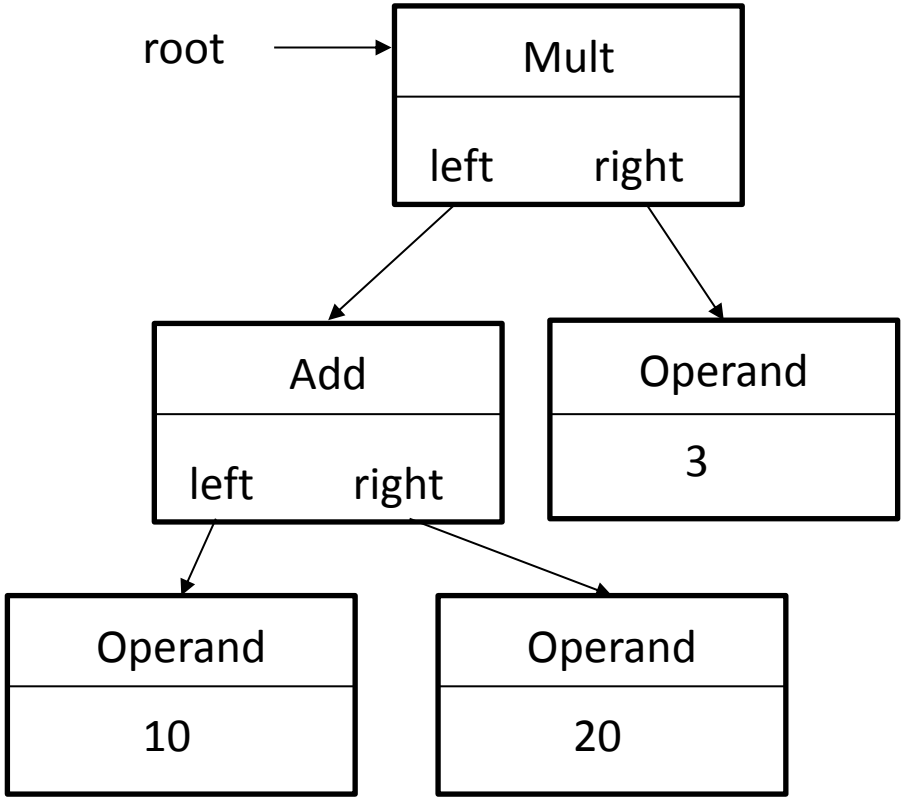
```
x = 1 y = 2
x = 1 y = 2 w = 1000 h = 2000
x = 1 y = 2 m = 100
[x = 1 y = 2, x = 1 y = 2 w = 1000 h = 2000, x = 1 y = 2 m = 100]
```

Esercizio: expression tree

Si tratta di una struttura che rappresenta le espressioni aritmetiche senza usare le parentesi per stabilire le precedenze.

L'espressione dell'esempio è:
(10 + 20) * 3.

Si vuole definire la struttura e come calcolare il risultato partendo dal nodo iniziale.

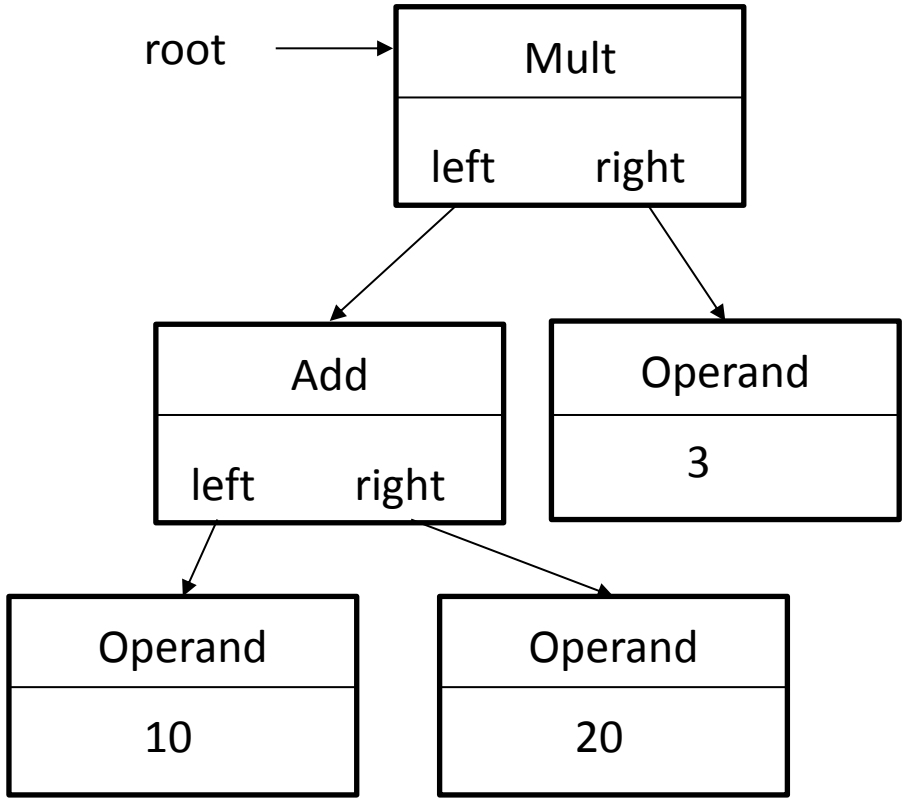


Analisi

Servono due tipi di nodi, operando e operatore.

Ad un operatore sono collegati due nodi il cui tipo può essere operando o operatore (serve quindi un'interfaccia che li accomuni).

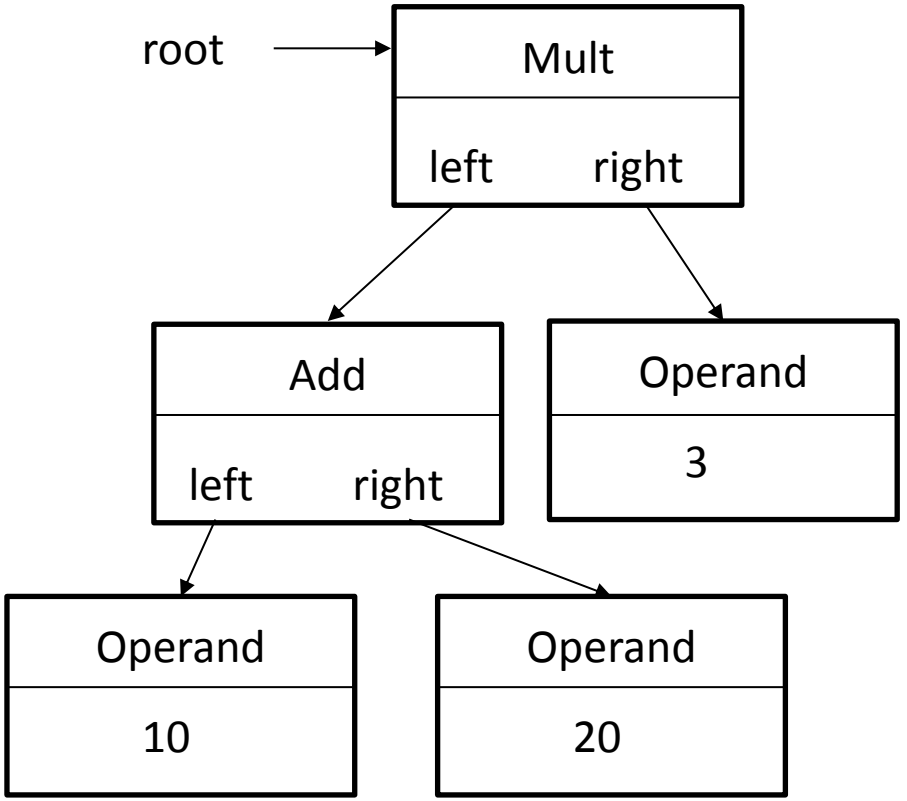
Gli operatori binari Add e Mult hanno due puntatori left e right.



L'interfaccia dei nodi:

```
public interface Computable {
int compute();
}
```

Scrittura dell'albero




```
public static void main(String[] args) {
Computable root = new Mult(
   new Add(new Operand(10), new Operand(20)),   new Operand(3));
System.out.println(root.compute());
}
```

```
public class Add implements Computable {
Computable left, right;
public int compute() {return left.compute() + right.compute();}
public Add(Computable left, Computable right)
        {this.left = left; this.right = right;}
}
public class Mult implements Computable {
Computable left, right;
public int compute() {return left.compute() * right.compute();}
public Mult(Computable left, Computable right)
        {this.left = left; this.right = right;}
}
public class Operand implements Computable {
int value;
public int compute() {return value;}
public Operand(int value){this.value = value;}
}
```

Implementazione

Nota

Si potrebbe definire la classe astratta **Operator** e rendere Add e Mult sue sottoclassi.

Operator definisce i due puntatori e pone il vincolo di implementare l'interfaccia.

```
public abstract class Operator implements Computable {
Computable left, right;
public abstract int compute();}


public class Add extends Operator{
public int compute() {return left.compute() + right.compute();}
public Add(Computable left, Computable right)
        {this.left = left; this.right = right;}
}
```

Estensione

Si vuole aggiungere il meno unario; ad es. – (10+20)* 3.

Analisi
Serve una nuova classe, Minus.
Si potrebbe definire una classe astratta per gli operatori unari se ne servissero altri, ad es. abs.

```
Computable root = new Minus (new Mult
(new Add(new Operand(10), new Operand(20)),  new Operand(3)));
System.out.println(root.compute());}
```

Class Minus

```
public class Minus implements Computable{
Computable item = null;
public int compute() {return -item.compute();}
public Minus (Computable item) {this.item = item;}
}
```

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 Movable {
        void move(int x, int y);
        int limit = 100;
        default boolean checkLimit(int x, int y) {
                return x <= limit && y <= limit;
        }
}
```

con checkLimit si controlla se le coordinate rientrano nel limite stabilito.

Nel programma di test si può aggiungere:

```
        System.out.println(m.checkLimit(120, 40)); // false
```

Metodi statici

InfoI ha due metodi abstract, un metodo default e un metodo statico.

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

Chiamata del metodo statico: InfoI.stampaListaInfoI

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 utilizzabili come parametri

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 Container <T> { ... }

Per usare questa classe occorre specializzarla con un tipo non generico

Es. Container <String> sc;

Container<Integer> ic;

Container <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.

Un tipo generico semplice

Si definisca un contenitore capace di contenere un solo oggetto, che si mette con il metodo put e si toglie con il metodo take.

```
public class Container <T> {
        private T contents = null;
        public void put (T item) {contents = item;} // stesso tipo T
        public T take() {T item = contents; contents = null; return item;}
}
```

Specializzazione per ottenere un contenitore di stringhe

```
Container <String> sc = new Container <String> ();
// sc opera soltanto su stringhe
sc.put("alfa");
System.out.println(sc.take()); // alfa
```

Analogamente si può definire un contenitore di Integer.

Il codice è scritto una volta sola nella classe generica.

La classe di libreria (in java.util) **Optional<T>** definisce un contenitore con maggiori funzionalità.

Classe e interfaccia generiche

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

```
public interface StackI <T> {
        boolean put (T t);
        T get ();
        int length ();
}
```

A differenza del contenitore lo stack contiene più oggetti; serve una struttura dati ad es. un array di Object.

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.

Progetto top-down: prima l'interfaccia poi l'implementazione

Stack<T>

```
package generici;
public class Stack <T> implements StackI <T> {
        private T[] stack;
        private int l = -1; // indice dell'ultimo elemento, -1 = vuoto

        @SuppressWarnings("unchecked")
        public Stack (int size) {stack = (T[]) new Object[size];}

        public boolean put (T t) {
                if (l == stack.length - 1) return false;
                else {stack[++l] = t; return true;}}

        public T get () {if (l == -1) return null; else return stack[l--];}

        public int length () {return l+1;}
}
```

Note

Lo stack è un array di Object perché T è ignoto. Lo stack è generato dal costruttore.

```
private T[] stack;
public Stack (int size) {
        stack = (T[]) new Object[size];
}
```

Nota: il cast da Object[] a T[] non garantisce che l'array contenga soltanto oggetti di tipo T (o derivati). Qui funziona perché inizialmente l'array contiene soltanto elementi null e inoltre ha un unico punto d'accesso (il rif. stack).

Senza la direttiva @SuppressWarnings("unchecked")

si avrebbe il warning

Type safety: Unchecked cast from Object[] to T[]

Programma di test

```
// due specializzazioni, Integer e String

public static void main(String[] args) {
Stack<Integer> intStack = new Stack<>(3);
Stack<String> stringStack = new Stack<>(4);
intStack.put(1); intStack.put(2); intStack.put(3); intStack.put(4);
System.out.println(intStack.get());
stringStack.put("alfa"); stringStack.put("beta");
System.out.println(stringStack.get());
System.out.println(stringStack.get());
System.out.println(stringStack.length());
}
```

```
3
beta
alfa
0
```

Iteratori: interfacce Iterable e Iterator

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.

L'iteratore rispetta l'interfaccia generica seguente (in java.util)

Interface Iterator<E>

boolean hasNext();

E next();

hasNext() dà true se ci sono altri elementi da visitare; E next() dà il prossimo elemento o lancia l'eccezione NoSuchElementException (RuntimeException) se non vi sono altri elementi.

L'iteratore è fornito dalla classe in modo standard attraverso l'interfaccia generica seguente (in java.lang), che è implementata dalla classe.

Interface Iterable<T>

Iterator<T> iterator()

Stack visitabile

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

forma compatta

```
for (String s:stringStack) System.out.println(s);
```

forma estesa

```
Iterator<Integer> iter = intStack.iterator();
while (iter.hasNext()) {
        int i = iter.next(); System.out.println(i);
}
```

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.

Nuova versione di Stack

```
package generici;
import java.util.*;
public class Stack <T> implements StackI <T>, Iterable <T> {
private T[] stack; private int l = -1;
// costruttore, put, get, length come prima
public Iterator<T> iterator () {return new StackIterator<T>();}

private class StackIterator<V> implements Iterator<V> {
        private int i = -1; // stato dell'iterazione
        public boolean hasNext() {return i < l;} // l è visibile
        @SuppressWarnings("unchecked")
        public V next() {
                if (!hasNext()) throw new NoSuchElementException();
                return (V) stack[++i];
        // senza direttiva: Type safety: Unchecked cast from T to V
}
}
```

2 interfacce

classe annidata

Note

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.

Digressione: 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.

```
public class EsempiArray2 {
    public static class Libro {
            private int codice; private String titolo;
            public Libro (int codice, String titolo) {this.codice = codice; this.titolo = titolo;}
            public int getCodice() {return codice;}
            public String getTitolo() {return titolo;}
    }
public static void main(String[] args) {
      Libro [] libri = {new Libro(1, "alfa"),new Libro(2, "beta"),
            new Libro(3, "delta"),new Libro(4, "gamma")};
      String titolo = "delta";
      for (Libro l:libri) if (l.getTitolo().equals(titolo))  {
            System.out.println(l.getCodice());
            break;
      }}
}
```

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)

Metodi generici

Scrivere un metodo statico che concateni due stack producendo un nuovo stack come risultato.

```
public class C {
public static <T>  Stack<T> unioneStack (Stack<T> s1, Stack<T> s2) {
        int l1 = s1.length(); int l2 = s2.length();
        Stack<T> s = new Stack<T> (l1+l2);
        for (T e:s1) s.put(e); for (T e:s2) s.put(e);
        return s;
}
}
```

Restrizioni applicabili ai parametri che sono tipi

Scrivere un metodo statico che sommi il contenuto di uno stack numerico.

```
public class C {
public static <T extends Number> double totale (Stack<T> s)  {
        double r = 0.0;
        for (Number n: s) r = r + n.doubleValue();
        return r;
}
}
```

<T **extends** Number**>** indica una classe che può essere Number o una sua sottoclasse

Variante

```
public class C {
public static double totale (Stack<? extends Number> s)  {
        double r = 0.0;
        for (Number n: s) r = r + n.doubleValue();
        return r;
}}
```

Confronto:

```
public static <T extends Number> double totale (Stack<T>> 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.

Programmi di test

```
public static void main(String[] args) { // in una classe diversa da C
Stack<Integer> intStack1 = new Stack<>(3);
        intStack1.put(1); intStack1.put(2);
Stack<Integer> intStack2 = new Stack<>(3);
        intStack.put(3); intStack.put(4);
Stack<Integer> intStack3 = C.unioneStack (intStack1, intStack2);

double totale = C.totale (intStack3);
}
```

Notare il diamond <>
che permette di lasciare sottinteso il tipo effettivo degli elementi.

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*.

Ordinamento naturale di stringhe

Ordinamento di un array di stringhe

```
String[] a = {"beta", "alfa", "gamma"};
Arrays.sort(a); // ordinamento alfabetico
System.out.println(Arrays.toString(a));//[alfa, beta, gamma]
```

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

String implementa Comparable<String> e il metodo

public **int compareTo** (String s)

dà -1, 0, oppure 1 a seconda che l'oggetto target

preceda (in ordine alfabetico) il parametro, sia uguale al parametro o lo segua.

Ordinamento naturale di Point

Se la classe è definita dall'utente occorre implementare l'interfaccia Comparable che contiene il metodo *compareTo*.

Qual è l'ordinamento naturale di Point?

Si può decidere che sia per x crescenti e a parità di x per y crescenti.

Ordinamento naturale di Point

```
package esempiOrdinamenti;
import java.util.*;
public class Point implements Comparable<Point> {
   private int x,y; …
   public int compareTo(Point p) {
         if (this.x == p.x)
                  if (this.y == p.y) return 0; else return this.y < p.y? -1:1;
         else return this.x < p.x? -1:1; }
}
```

Esempio di test

```
Point[] v = {new Point(5,30), new Point(5,20), new Point(3,20)};
Arrays.sort(v); System.out.println(Arrays.toString(v));
```

[x = 3 y = 20, x = 5 y = 20, x = 5 y = 30]

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.

Nota: un comparatore di T o di una superclasse va bene (Lower Bounded Wildcard).

Si può definire un comparatore in 3 modi:
con una classe normale,
con una classe anonima,
con una lambda expression.

Ordinamento in senso inverso di un array di stringhe: classe normale

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

programma di test
String[] a1 = {"beta", "alfa", "gamma"};
StringComparator comp1 = new StringComparator();
Arrays.sort(a1, comp1);
System.out.println(Arrays.toString(a1)); //[gamma, beta, alfa]
```

La classe contiene un unico metodo e il costruttore è quello di default.

Esempio con classe anonima

Si può inglobare la classe (come classe anonima) nel metodo che effettua l'ordinamento, evitando così di definire una classe separata.

```
String[] a2 = {"beta", "alfa", "gamma"};
Comparator<String> comp2 = new Comparator<String>() {
        public int compare(String s1, String s2)
                {return s2.compareTo(s1);}
};
Arrays.sort(a2,comp2);
System.out.println(Arrays.toString(a2)); //[gamma, beta, alfa]
```

La classe è anonima perché è indicata dall'interfaccia che deve implementare.

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 λ 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).

Test di esecuzione della λ:

int i = comp.compare("alfa", "beta"); // i = 1 quindi alfa segue beta (scambio)

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

Sorting con λ

```
String[] a3 = {"beta", "alfa", "gamma"};
Comparator<String> comp = (s1, s2) -> s2.compareTo(s1);
Arrays.sort(a3, comp);
System.out.println(Arrays.toString(a3)); //[gamma, beta, alfa]
```

Oppure

```
String[] a3 = {"beta", "alfa", "gamma"};
Arrays.sort(a3, (s1, s2) -> s2.compareTo(s1));
```

Si passa la λ alla sort come secondo parametro.

Struttura di una lambda expression

parametri -> body

parametri

nessuno: ()

uno: x oppure (Tipo x) dove Tipo è il tipo di x

due o più: (x, y) oppure (Tipo1 x, Tipo2 y)

body

espressione:
```
System.out.println("Hello world")
x.getValue() > 100
x + y
```

blocco: tra graffe con eventuale return
```
{System.out.print("Hello ");
        System.out.println("world");}
{return x.getValue() > 100;}
{return x + y;}
```

Interfacce funzionali di uso comune

Si trovano nel package java.util.function:

Function <T,R>	R apply(T t)	T → R
BiFunction<T,U,R>	R apply(T t, U u)	T, U → R
BinaryOperator <T>	T apply(T t, T u)	T, T → T
	parametri e risultato dello stesso tipo	
UnaryOperator<T>	T apply(T t)	T → T
Predicate <T>	boolean test(T t)	T → boolean
Consumer <T>	void accept(T t)	T → void
BiConsumer<T,U>	void accept(T t, U u)	T, U → void
Supplier <T>	T get()	void → T

Altre interfacce

ToIntFunction<T> int applyAsInt(T value) T → int

analogamente per

ToLongFunction<T>

ToDoubleFunction<T>

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

Si può anche usare un *method reference (*λ con 1 parametro)

f1 = String::toUpperCase; //method reference
r = f1.apply("alfa");
System.out.println(r); //ALFA

Method reference: nome classe :: nome metodo
x -> x.toUpperCase()   diventa String::toUpperCase

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;

Se vogliamo ottenere il valore dell'attributo x di un Point, dobbiamo implementare la Function<Point,Integer> e il method reference è

Point::getX

Function<Point,Integer> g = Point::getX;

oppure

Function<Point,Integer> g = p -> p.getX();

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*.

BinaryOperator <T>

```
BinaryOperator <Integer> somma = (a, b) -> a + b;
int k = somma.apply(10, 20);
System.out.println(k); //30

BinaryOperator <Integer> moltiplicazione = (a, b) -> a * b;
k = moltiplicazione.apply(somma.apply(10,20), 3);
System.out.println(k); //90
```

BinaryOperator <T>

Contiene anche

static <T> BinaryOperator<T> **maxBy** (Comparator<? super T> comparator)

static <T> BinaryOperator<T> **minBy** (Comparator<? super T> comparator)

Sono metodi statici.

Forniscono un operatore binario che mediante il comparatore ricevuto dà il maggiore o il minore dei due parametri.

Nota: il comparatore concreto può anche essere di una superclasse di T:
<? super T> indica la classe T o una qualsiasi superclasse di T (Lower Bounded Wildcard).

Nota: occorre indicare che T è un type parameter, altrimenti la scrittura

? super T

non è valida.

Esempi di maxBy

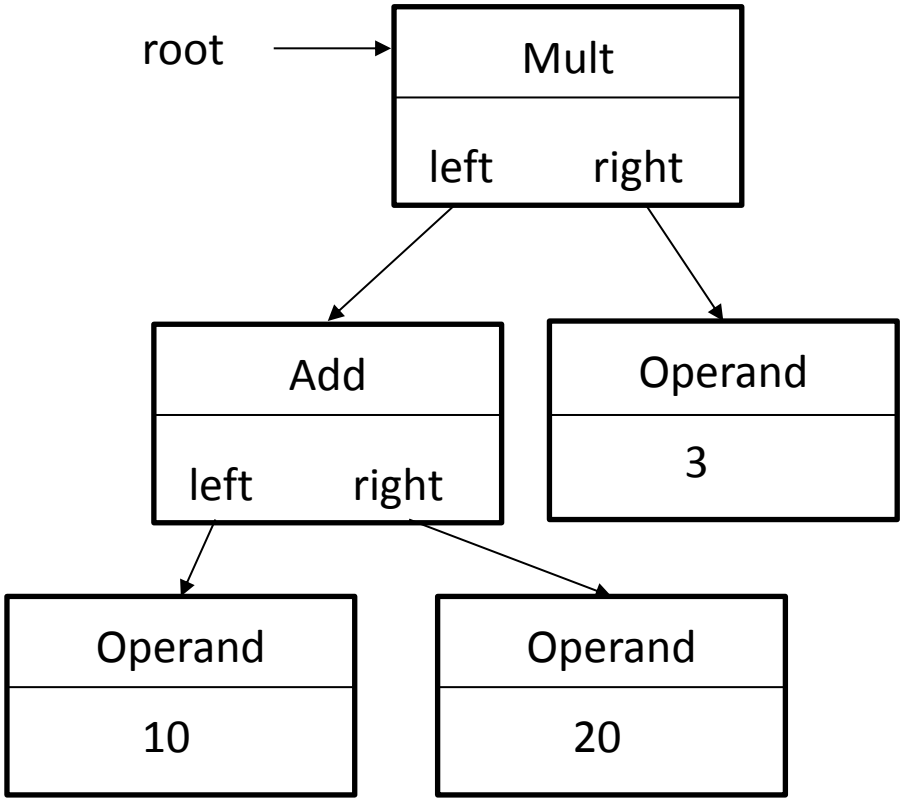
```
Comparator<Integer> comp = (x, y) -> {return x == y? 0: x < y? -1:1;};
BinaryOperator<Integer> maggiore= BinaryOperator.maxBy(comp);

int nMaggiore = maggiore.apply(10,20); //20
//oppure
Integer intMaggiore = maggiore.apply(10,20);
```

Esercizio: expression tree con lambda expressions

Servono due tipi di nodi, operando e operatore.

Ad un operatore sono collegati due nodi il cui tipo può essere operando o operatore (serve quindi un'interfaccia che li accomuni).

Gli operatori binari Add e Mult hanno due puntatori left e right.



L'interfaccia dei nodi:

```
public interface Computable {
int compute();
}
```

Con lambda expressions

Basta un'unica classe per tutti gli operatori binari; il comportamento è dato da una BiFunction passata al costruttore come parametro.

```
public class BinaryOpr implements Computable{
        Computable left, right;
        BiFunction<Computable,Computable,Integer> f;
        //lambda come attributo d'istanza
        public int compute() {return f.apply(left, right);}
        public BinaryOpr(Computable left, Computable right,
                BiFunction<Computable,Computable,Integer> f)
                // lambda come parametro
                {this.left = left; this.right = right; this.f = f;}
}
```

Con lambda expressions

```
BiFunction<Computable,Computable,Integer> somma =
        (x,y) -> x.compute() + y.compute();
BiFunction<Computable,Computable,Integer> moltiplicazione =
        (x,y) -> x.compute() * y.compute();

        Computable root = new BinaryOpr(
        new BinaryOpr(new Operand(10), new Operand(20), somma),
        new Operand(3), moltiplicazione);
        System.out.println(root.compute());
```

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

Interfaccia Comparator <T>

Ad es. si supponga di voler ordinare una collezione di punti per x decrescenti e a parità di x per y decrescenti.

Con una λ si ha:

```
Comparator<Point> comp1 =
        (p1,p2) -> {
                if (p1.getX() == p2.getX())
                        if (p1.getY() == p2.getY()) return 0;
                        else return (p1.getY() < p2.getY()? 1 : -1);
                else return (p1.getX() < p2.getX()? 1 : -1);} ;
```

Comparatore di punti mediante comparing

I punti sono ordinati per x decrescenti e a parità di x per y decrescenti.

Si usano comparing e thenComparing

```
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.

Nell'esempio, la sort key è l'attributo x e il comparatore confronta le x secondo l'inverso del criterio naturale (quindi ordina per x decrescenti).

Il thenComparing entra in azione quando il primo comparatore trova due x uguali.

Esempio completo

```
import static java.util.Comparator.*;
import java.util.*;
public class EsempioComparing {
public static void main(String[] args)  { //x decresc y decresc

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

Point[] v = {new Point(5,40), new Point(10,20)};
Arrays.sort(v, comp1);
System.out.println(Arrays.toString(v));
}}
```

[x = 10 y = 20, x = 5 y = 40]

import static

```
import static java.util.Comparator.*;
import java.util.*;
public class EsempioComparing {
public static void main(String[] args)  { //x decresc y decresc

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

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

Nell'esempio è servito per comparing, thenComparing e reverseOrder.

Variante

```
Point[] v = {new Point(5,40), new Point(10,20)};

Arrays.sort(v, comparing(Point::getX, reverseOrder())
            .thenComparing(Point::getY, reverseOrder()) );
System.out.println(Arrays.toString(v));
}}
```

Si può passare il comparatore prodotto da comparing e thenComparing direttamente al metodo sort come secondo parametro. Tuttavia data la complessità della scrittura risultante conviene limitarsi a casi semplici come quello della slide seguente.

Variante

```
// ordinamento in senso inverso di un array di stringhe
String[] a4 = {"beta", "alfa", "gamma"};

Arrays.sort(a4, comparing(s -> s, reverseOrder()));

System.out.println(Arrays.toString(a4));//[gamma, beta, alfa]
```

L'ordinamento è su tutto l'oggetto (stringa) e non su un attributo; si usa quindi la λ s -> s.

Approfondimento: dichiarazioni di comparing (1)

1 parametro: ordinamento naturale sulla sort key

Esempio: Comparator<Point> c = comparing(Point::getX);

T è Point, U è Integer.

static <T,U extends Comparable <? super U>> Comparator<T>
comparing (Function<? super T, ? extends U> **keyExtractor**);

T è il tipo (classe) degli oggetti da confrontare e U è il tipo della sort key.

Il risultato è un Comparator<T>.

U deve implementare l'interfaccia Comparable<U> oppure Comparable di una superclasse di U. La notazione <? super U> indica U oppure una superclasse di U.

Il key extractor può operare su T o su una sua superclasse <? super T>. Il tipo del risultato è U o una sottoclasse di U.

Approfondimento: dichiarazioni di comparing (2)

2 parametri: il secondo specifica l'ordinamento

static <T,U> Comparator <T>

comparing (Function<? super T,? extends U> **keyExtractor**,

Comparator<? super U> **keyComparator**);

keyComparator deve implementare Comparator di U o di una superclasse di U.

Dichiarazioni di comparing (2)

Varianti di comparing: per Integer, Long e Double

static <T> Comparator<T> comparingInt (ToIntFunction<? super T> keyExtractor);

static <T> Comparator<T> comparingLong (ToLongFunction<? super T> keyExtractor);

static <T> Comparator<T> comparingDouble (ToDoubleFunction<? super T> keyExtractor);

Dichiarazioni di thenComparing

1 parametro: la chiave con ord. naturale in base alle chiavi

default <U extends Comparable<? super U>> Comparator<T> **thenComparing** (Function<? super T,? extends U> **keyExtractor**);

2 parametri: la chiave e il comparatore per un ord. specifico in base alle chiavi

default <U> Comparator<T> **thenComparing** (Function<? super T,? extends U> **keyExtractor**, Comparator<? super U> **keyComparator**); // ord. definito

1 parametro: il comparatore per un ord. degli oggetti

default Comparator<T> **thenComparing** (Comparator<? super T> other)
// confronta gli oggetti di tipo T in base al comparatore fornito
// es: comparing(String::length).thenComparing(naturalOrder());

Dichiarazioni di thenComparing

Varianti per Integer, Long e Double

default Comparator<T> thenComparingInt (ToIntFunction<? super T> keyExtractor);

default Comparator<T> thenComparingLong(ToLongFunction<? super T> keyExtractor);

default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor);

Ordinamenti

ordinamento naturale

static <T extends Comparable<? super T>> Comparator<T>
naturalOrder()

ordinamento inverso a quello naturale

static <T extends Comparable<? super T>> Comparator<T>
reverseOrder()

default Comparator<T> **reversed**()

dà un comparatore che inverte l'ordinamento del comparatore a cui è applicato.

Esempio:ordinamento di un array di punti

```
Point[] v = {new Point(5,40), new Point(10,20), new Point(10,30)};
Comparator<Point> comp1 = // ord per x e y decrescenti
        comparing(Point::getX, reverseOrder())
        .thenComparing(Point::getY, reverseOrder());
Arrays.sort(v, comp1);  // [x = 10 y = 30, x = 10 y = 20, x = 5 y = 40]
                System.out.println(Arrays.toString(v));

Comparator<Point> comp2 = // ord naturale
        comparing(Point::getX)
        .thenComparing(Point::getY, naturalOrder());
Arrays.sort(v, comp2); // [x = 5 y = 40, x = 10 y = 20, x = 10 y = 30]

Comparator<Point> comp3 = comp2.reversed();
Arrays.sort(v, comp3); // [x = 10 y = 30, x = 10 y = 20, x = 5 y = 40]
```

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

Interfacce

Collection

- Set
  - SortedSet
- List
- Queue
  - Deque

Map

- SortedMap

La rientranza indica inheritance.

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)

es. c1.addAll(c2)

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

es. c1.removeAll(c2)

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

es. c1.retainAll(c2)

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.

String[] elenco = {"beta", "omega", "alfa", "beta", "alfa", "delta", "gamma", "alfa"};

Risposte

parole: [alfa, beta, delta, gamma, omega]

parole ripetute: [alfa, beta]

paroleNonRipetute: [delta, gamma, omega]

prima parola: alfa

toString di set presenta il toString degli elementi separati da ", " e racchiusi tra [].

Procedimento

String[] elenco = {"beta", "omega", "alfa", "beta", "alfa", "delta", "gamma", "alfa"};

Dall'array dato si costruiscono due set: quello (ordinato) delle parole (parole) e quello delle parole ripetute (parole ripetute).

Per ottenere le parole non ripetute si copia il primo set in un terzo set (paroleNonRipetute) e si sottraggono da questo le parole del secondo set.

Servono un treeSet e due hashSet.

Esempio

```
SortedSet<String> parole = new TreeSet<String>();
Set<String> paroleRipetute = new HashSet<String>();
for (String s:elenco) if (!parole.add(s)) paroleRipetute.add(s);

System.out.println(parole); // [alfa, beta, delta, gamma, omega]
System.out.println(paroleRipetute); // [alfa, beta]
System.out.println(parole.first());  // alfa

HashSet<String> paroleNonRipetute = new HashSet<>(parole);
        // copia di parole
paroleNonRipetute.removeAll(paroleRipetute);
    System.out.println(paroleNonRipetute); // [delta, gamma, omega]
```

Note

SortedSet<String> **parole** = new TreeSet<String>();
Set<String> **paroleRipetute** = new HashSet<String>();

Si può anche scrivere

TreeSet<String> **parole** = new TreeSet<String>();
HashSet<String> **paroleRipetute** = new HashSet<String>();

HashSet<> equivale a HashSet<String> per *type inference*.

toString di set presenta il toString degli elementi separati da ", " e racchiusi tra [].

Esempi di toArray e removeIf

```
System.out.println(parole); // [alfa, beta, delta, gamma, omega]
Object[] parole1 = parole.toArray();

String[] parole2  = new String[parole.size()];
parole.toArray(parole2);
```

Metodo removeIf: rimozione di omega

```
boolean removeIf (Predicate<? super E> filter)

System.out.println(parole); // [alfa, beta, delta, gamma, omega]
HashSet<String> parole3 = new HashSet<>(parole); // copia di parole
parole3.removeIf(p -> p.equals("omega"));
System.out.println(parole3); // [delta, alfa, beta, gamma]
```

Costruttori di HashSet <E>

HashSet()

vuoto con capacità iniziale 16 e load factor 0.75.

HashSet(Collection<? extends E> c)

con gli elementi distinti di c.

HashSet(int initialCapacity)

HashSet(int initialCapacity, float loadFactor)

Costruttori di TreeSet <E>

TreeSet()

vuoto con ordinamento naturale.

TreeSet(Collection<? **extends** E> c)

con gli elementi distinti di c e ordinamento naturale.

TreeSet(Comparator<? **super** E> comparator)

vuoto con ordinamento esterno.

TreeSet(SortedSet<E> s)

con gli elementi e l'ordinamento di s

TreeSet con comparatore

Da un elenco di parole si vuole ottenere un set ordinato per lunghezze crescenti e alfabeticamente.

```
Comparator<String> comp =
  comparing(String::length).thenComparing(naturalOrder());

SortedSet<String> parole = new TreeSet<String>(comp);
String[] v = {"tau", "omega", "alfa", "beta", "eta", "delta", "omicron"};
for (String s:v) parole.add(s);
System.out.println(parole);  // [eta, tau, alfa, beta, delta, omega, omicron]
```

Che succede se si omette il thenComparing?

[tau, alfa, omega, omicron]

tau ed eta sono trattate come uguali dato che il confronto è fatto sulla lunghezza quindi eta è scartata; idem per alfa e beta, omega e delta.

Set con nuovi tipi (non predefiniti)

HashSet

la classe deve implementare i metodi hashCode e equals

Un HashSet aggiunge un nuovo elemento se:

il suo hashCode è diverso da quelli già presenti oppure

se equals è falso per tutti gli oggetti che hanno lo stesso hashCode.

TreeSet

ordinamento naturale: *la classe deve implementare* l'interfaccia **Comparable** (e quindi il metodo **compareTo**);

ordinamento esterno: occorre fornire al costruttore del treeSet un oggetto (*comparatore*) che implementi l'interfaccia **Comparator** (e quindi il metodo **Compare**).

Le classi di libreria ridefiniscono *hashCode, equals* e implementano l'interfaccia *Comparable*.

Esempio: Point come classe di libreria

```
public class Point implements Comparable<Point> {
    private int x,y;
    public int getX() {return x;}
    public int getY() {return y;}
    public Point(int x, int y) {this.x = x; this.y = y;}
    public String toString(){return "(" + x + "," + y+ ")";}
```

Con le implementazioni di equals, hashCode e compareTo può essere usata con tutti i tipi di set.

```
    public boolean equals (Object o) {
        if (!(o instanceof Point)) return false;
        Point p = (Point) o; return x == p.x && y == p.y;}
    public int hashCode () {return x + 31 * y;}
    public int compareTo (Point p) { // increasing x incr y
        if (x == p.x) {
            if (y == p.y) return 0;
            return y < p.y ? -1: 1;
        } else return x < p.x ? -1: 1;
    }
}
```

Programma di test

```
Set<Point> ps = new HashSet<>();
ps.add(new Point(10,20)); ps.add(new Point(15,25));
ps.add(new Point(5,25));
ps.add(new Point(10,20)); // add non eseguita perché il punto è duplicato
ps.add(new Point(15,30));
                                System.out.println(ps);
```

```
[(15,30), (10,20), (15,25), (5,25)]

[(5,25), (10,20), (15,25), (15,30)]
```

```
Set<Point> ps1 = new TreeSet<Point>(ps);
                                System.out.println(ps1); // ord per x e y crescenti
```

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.

Costruttori di ArrayList

ArrayList()

vuota con capacità iniziale di 10.

ArrayList(Collection<? extends E> c)

ArrayList(int initialCapacity)

Interfaccia ListIterator<E>

E next() dà l'elemento successivo o lancia NoSuchElementException

int nextIndex() dà l'indice dell'elemento successivo o list size

E previous() dà l'elemento precedente o lancia NoSuchElementException

int previousIndex() dà l'indice dell'elemento precedente o -1

boolean hasNext()

boolean hasPrevious()

void add(E e) inserisce prima dell'elemento dato da next o dopo l'elemento dato da previous

void remove() rimuove l'ultimo elemento dato da next o previous

void set(E e) sostituisce l'ultimo elemento dato da next o previous con l'elemento e

Esempi di operazioni su liste

add, toString, get, indexOf, set

```
List<String> list = new ArrayList<>();
list.add("alfa"); list.add("gamma"); list.add("beta"); list.add(null);
list.add(2,"omega");
System.out.println(list); // [alfa, gamma, omega, beta, null]
System.out.println(list.get(2));  // omega
System.out.println(list.indexOf("omega")); // 2
list.set(2, "epsilon");  // epsilon al posto di omega
System.out.println(list); // [alfa, gamma, epsilon, beta, null]
```

Esempi di operazioni su liste

addAll, costruttore di copia, remove

List<String> list1 = new ArrayList<>(); list1.**addAll**(list);
oppure List<String> list1 = new ArrayList<>**(list)**;

list1.**remove**("beta"); list1.remove(0);
System.out.println(list1); // [gamma, epsilon, null]

Note

Una lista può contenere dei null.

toString di lista presenta il toString degli elementi separati da ", " e racchiusi tra [].

Esempi di operazioni su liste

Eliminazione dei null con iteratore

```
//list = [alfa, gamma, epsilon, beta, null]

Iterator<String> iter = list.iterator();
while (iter.hasNext()) if (iter.next()==null) iter.remove();
System.out.println(list); // [alfa, gamma, epsilon, beta]
```

Eliminazione dei null con removeIf

```
list.removeIf(s -> s == null);
```

swap

```
Collections.swap(list,0,1);  System.out.println(list);
// [gamma, alfa, epsilon, beta] scambio tra alfa e gamma
```

Esempi di operazioni su liste

Iterazione all'indietro

```
// list = [gamma, alfa, epsilon, beta]

ListIterator<String>listIter = list.listIterator(list.size());
while (listIter.hasPrevious()) System.out.println(listIter.previous());
```

```
beta
epsilon
alfa
gamma
```

subList

```
System.out.println(list.subList(0,3)); //[gamma, alfa, epsilon]
```

Esempi di operazioni su liste

binarySearch

```
//list = [alfa, beta, epsilon, gamma]
System.out.println(Collections.binarySearch(list, "epsilon")); // 2
System.out.println(Collections.binarySearch(list, "delta")); // -3
//delta manca, dovrebbe essere inserito in posiz. 2 (-key -1)
```

reverse

```
Collections.reverse(list); System.out.println(list);
//[gamma, epsilon, beta, alfa]
```

Ordinamenti

naturale

//[gamma, epsilon, beta, alfa]

Collections.**sort**(list);
System.out.println(list); // [alfa, beta, epsilon, gamma]

con comparatore

ad es. per lunghezza crescente e alfabeticamente a parità di lunghezza
list.add("eta"); // [alfa, beta, epsilon, gamma, eta]

```
Comparator<String> comp =
   comparing(String::length).thenComparing(naturalOrder());

Collections.sort(list, comp);  //[eta, alfa, beta, gamma, epsilon]
```

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.

Costruttori di LinkedList

LinkedList()

LinkedList(Collection<? extends E> c)

Costruttori di PriorityQueue

PriorityQueue() // ordinamento naturale

PriorityQueue(Collection<? extends E> c)

PriorityQueue(Comparator<? super E> comparator)

PriorityQueue(int initialCapacity)

PriorityQueue(int initialCapacity, Comparator<? super E> comparator)

PriorityQueue(PriorityQueue<? extends E> c)

PriorityQueue(SortedSet<? extends E> c)

Esempi di LinkedList

costruttore di copia, addFirst, addLast, peek

```
List<String> list = new ArrayList<>();
list.add("alfa"); list.add("gamma"); list.add("beta"); list.add(2,"omega");
LinkedList<String> linkedList = new LinkedList<>(list);

linkedList.addFirst("lambda"); linkedList.addLast("zeta");
System.out.println(linkedList);
//[lambda, alfa, gamma, omega, beta, zeta]
System.out.println(linkedList.peek()); // lambda
```

Esempi di PriorityQueue

```
//list = [alfa, gamma, omega, beta]

PriorityQueue<String> priorityQ = new PriorityQueue<>(list);
System.out.println(priorityQ);
//[alfa, beta, omega, gamma]  è ordinata ma non si vede
priorityQ.add("delta");
int n = priorityQ.size();
for (int i = 0; i < n; i++)
        System.out.println(priorityQ.remove());
//l'estrazione è ordinata
```

alfa
beta
delta
gamma
omega

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*.

Costruttori

HashMap() come HashSet
HashMap(int initialCapacity)
HashMap(int initialCapacity, float loadFactor)
HashMap(Map<? extends K,? extends V> m)

TreeMap() come TreeSet, ordinamento naturale
TreeMap(Comparator<? super K> comparator)
TreeMap(Map<? extends K,? extends V> m)
    copia con ordinamento naturale
TreeMap(SortedMap<K,? extends V> m)
    copia con ordinamento della mappa m.

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)

SortedMap

Alcuni metodi

K firstKey()
K lastKey()
SortedMap<K,V> headMap(K toKey)
//vista limitata alle chiavi < toKey
SortedMap<K,V> tailMap(K fromKey)
//vista limitata alle chiavi >= fromKey
SortedMap<K,V> subMap(K fromKey, K toKey)
//vista con chiavi nel range fromKey (inclusa) – toKey (esclusa)

Esempio: si contino le occorrenze delle parole in un testo

Esempio

String testo = "bianco *rosso* blu *verde* grigio bianco nero *verde* giallo *rosso*"; // in corsivo le parole duplicate

Ordinamento per parole (alfabetico)

[bianco:2, blu:1, giallo:1, grigio:1, nero:1, rosso:2, verde:2]

Ordinamento per occorrenze decrescenti e alfabetico

[bianco:2, rosso:2, verde:2, blu:1, giallo:1, grigio:1, nero:1]

Raggruppamento per lunghezza crescente delle parole in ordine alfabetico (e distinte)

{3=[blu], 4=[nero], 5=[rosso, verde], 6=[bianco, giallo, grigio]}

Nota: **toString di mappa** presenta i toString delle coppie chiave-valore separati da ", " e racchiusi tra {}; chiave e valore sono separati da =.

.

Soluzione

Che cos'è un valore in questo caso?

Un'occorrenza, cioè un oggetto che contiene la parola come id e il n. di occorrenze nel testo.

```
public  class Occorrenza {
        private String parola; private int n = 1;
        public Occorrenza(String parola) {this.parola = parola;}

        public String toString(){return parola + ":" + n;}
        public void incr() {n++;}

        public String getParola() {return parola;}
        public int getN() {return n;}
}
```

Programma di test

String testo = "bianco rosso blu verde grigio bianco nero verde giallo rosso";

Costruzione della mappa iniziale *mappa*

```
String[] parole = testo.split(" ");
Map<String, Occorrenza> mappa = new TreeMap <>();
for (String s: parole) {
        Occorrenza o = mappa.get(s);
        if (o == null) mappa.put(s, new Occorrenza(s)); else o.incr();
}
```

Programma di test

L'ordinamento alfabetico delle occorrenze per parole è automatico perché le chiavi stanno in un treeSet e quindi la collezione delle occorrenze è già ordinata

System.out.println(**mappa.values**());

[bianco:2, blu:1, giallo:1, grigio:1, nero:1, rosso:2, verde:2]

Programma di test

Per avere l'ordinamento per occorrenze decrescenti e alfabetico occorre riordinare la collezione dei valori mediante una lista.

I valori sono inseriti in una lista con il costruttore di copia

```
List<Occorrenza> occorrenze = new ArrayList<>(mappa.values());
```

Si definisce il comparatore

```
Comparator<Occorrenza> comp = comparing(Occorrenza::getN,
        reverseOrder()).thenComparing(Occorrenza::getParola);
```

Si ordina la lista

Collections.sort(occorrenze, comp); System.out.println(occorrenze);

[bianco:2, rosso:2, verde:2, blu:1, giallo:1, grigio:1, nero:1]

Programma di test

Per il raggruppamento per lunghezza crescente delle parole distinte in ordine alfabetico occorre costruire una nuova mappa con caratteristiche di multimap. Non serve la classe Occorrenza.

Una multimap ha per valori delle collezioni; in questo caso un treeSet delle parole (senza duplicati) che hanno la stessa lunghezza.

```
SortedMap<Integer,TreeSet<String>> m = new TreeMap<>();
for (String s: parole) {
   TreeSet<String> gruppo = m.get(s.length());
    if (gruppo == null) { // prima parola con questa lunghezza
            gruppo = new TreeSet<String>(); m.put(s.length(), gruppo);
    }
   gruppo.add(s);
}
System.out.println(m);
// {3=[blu], 4=[nero], 5=[rosso, verde], 6=[bianco, giallo, grigio]}
```

Esempio di foreach

```
m.forEach((k,v) -> System.out.println(k + " " + v));
```

```
3 [blu]
4 [nero]
5 [rosso, verde]
6 [bianco, giallo, grigio]
```

La lambda expression (BiConsumer) è

(k,v) -> System.out.println(k + " " + v)

Quali sono i tipi di k e v?

Type inference: dalla definizione della multimap → Integer e TreeSet.

Raggruppamento con stream

In questo caso lo stream è la sequenza di stringhe contenute nell'array parole.

```
Map<Integer,List<String>> m1 = Arrays.stream(parole)
.distinct() // filtro che elimina i duplicati usando equals
.sorted()  // ordinamento naturale
.collect(groupingBy(String::length, toList()));
//operazione finale: raggruppa le stringhe (in liste) in base alla lunghezza
System.out.println(m1);
```

```
{3=[blu], 4=[nero], 5=[rosso, verde], 6=[bianco, giallo, grigio]}
```

Si deve includere
import static java.util.stream.Collectors.*;

metodo di Arrays
static <T> Stream<T> stream(T[] array)

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.

Gestione ordini

La classe GestioneOrdini ha due metodi principali.

Il metodo

void **addProdotto** (String nome, int prezzo) throws Exception

inserisce un nuovo prodotto o lancia un'eccezione se esiste già un prodotto con quel nome.

Il metodo

Fornitore **addFornitore** (String nome, String… nomiProdotti) throws Exception

inserisce un nuovo fornitore o lancia un'eccezione se esiste già un fornitore con quel nome.

Se un nome di prodotto corrisponde ad un prodotto inserito con il metodo precedente, aggiunge il prodotto alla lista dei prodotti trattati dal fornitore.

Programma di test

```
GestioneOrdini g = new GestioneOrdini();
g.addProdotto("divano2p", 200); g.addProdotto("poltrona", 150);
g.addProdotto("divano2p", 300);
Fornitore f = g.addFornitore("AlfaMobili", "divano2p", "poltrona",
"libreria");
System.out.println(f);
```

Exception in thread "main" java.lang.Exception:
prodotto duplicato divano2p

```
GestioneOrdini g = new GestioneOrdini();
g.addProdotto("divano2p", 200); g.addProdotto("poltrona", 150);
//g.addProdotto("divano2p", 300);
Fornitore f = g.addFornitore("AlfaMobili", "divano2p", "poltrona",
"libreria");
System.out.println(f);  //AlfaMobili [divano2p 200, poltrona 150]
// la libreria non compare tra i prodotti del fornitore
```

Classi Prodotto e Fornitore

```
public class Prodotto {
private String nome; private int prezzo;
public Prodotto(String nome, int prezzo) {
        this.nome = nome; this.prezzo = prezzo;}
public String toString() {return nome + " " + prezzo;}
}

public class Fornitore {
private String nome;
private ArrayList<Prodotto> listaProdotti = new ArrayList<>();
public Fornitore(String nome) {this.nome = nome;}
public void addProdotto(Prodotto p) {listaProdotti.add(p);}
public String toString() {return nome + " " + listaProdotti;}
}
```

Classe GestioneOrdini

```
public class GestioneOrdini {
private Map<String, Prodotto> prodotti = new HashMap<>();
private Map<String, Fornitore> fornitori = new HashMap<>();

public void addProdotto (String nome, int prezzo) throws Exception {
if (prodotti.containsKey(nome))
        throw new Exception("prodotto duplicato " + nome);
prodotti.put(nome, new Prodotto(nome, prezzo));
}
```

controllo della duplicazione

```
if (prodotti.containsKey(nome))
```

Classe GestioneOrdini

```
public class GestioneOrdini {
private Map<String, Prodotto> prodotti = new HashMap<>();
private Map<String, Fornitore> fornitori = new HashMap<>();


public Fornitore addFornitore (String nome, String... nomiProdotti)
throws Exception {
if (fornitori.containsKey(nome))
        throw new Exception("fornitore duplicato " + nome);
Fornitore f = new Fornitore (nome);
for (String nomeProdotto: nomiProdotti) {
        Prodotto p = prodotti.get(nomeProdotto);
        if (p != null) f.addProdotto(p);
}
return f;}
}
```

Controllo della presenza o assenza; invece di ignorare l'assenza il programma potrebbe lanciare un'eccezione

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>.

Esempi di sorgenti di uno stream

IntStream.of (1, 3, 5, 7)

Stream<String>.of ("alfa", "beta", "gamma")

Data la classe Libro e la lista ArrayList<Libro> libri
libri.stream()

Dato l'array String[] parole

Arrays.stream(parole) dà uno Stream<String>

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.

Stream<String>

Esempio

```
Map<Integer, List<String>> m1 = Arrays.stream(parole)
.distinct() // filtro che elimina i duplicati usando equals
.sorted()  // ordinamento naturale
.collect(groupingBy(String::length, toList()));
```

Map<Integer, List<String>> è il tipo del risultato

m1 raggruppa le stringhe (in liste) in base alla lunghezza

```
{3=[blu], 4=[nero], 5=[rosso, verde], 6=[bianco, giallo, grigio]}
```

Funzione sorgente usata negli esempi

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

Stream<String> getStream() {
return Stream.of("bianco", "rosso", "blu", "verde", "grigio", "bianco",
"nero", "verde", "giallo", "rosso"); } // //bianco, rosso e verde sono duplicati
```

package usati negli esempi

Questa funzione serve per generare lo stesso stream negli esempi seguenti.

Operazione iniziale di Stream<T>

iniziale

static <T> Stream<T> **of** (T... values)

Operazioni intermedie di Stream<T>

Stream<T> **distinct** () // usa equals
Stream<T> **filter** (Predicate<? super T> predicate)
Stream<T> **sorted** () // ord. naturale
Stream<T> **sorted** (Comparator<? super T> comparator) //ord. esterno

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

Altre operazioni intermedie di Stream<T>

Stream<T> **peek** (Consumer<? super T> action)

//esegue l'azione per ogni elemento, utile per debugging

Stream<T> **limit** (long maxSize)

//se lo stream ha lunghezza > maxSize, la parte restante è ignorata

Stream<T> **skip** (long n)

//scarta i primi n elementi dello stream

Operazioni finali principali di Stream<T>

long **count** (): conta il numero degli elementi

Optional<T> **max** (Comparator<? super T> comparator)
Optional<T> **min** (Comparator<? super T> comparator)
dà l'oggetto maggiore o minore secondo il comparatore

Object[] **toArray** (): dà l'array contenente la sequenza finale

<R,A> R **collect** (Collector<? super T,A,R> collector)
dà il risultato con un collettore

void **forEach** (Consumer<? super T> action)
esegue un'elaborazione per ogni elemento della sequenza finale

IntStream

Ha metodi simili a quelli di Stream<T>

In più
OptionalDouble **average**( )
int **sum** ()
OptionalInt max()
OptionalInt min()

Analogamente per gli altri stream numerici LongStream, DoubleStream.

Esempio

Esempio: fornire un array con le stringhe distinte, di lunghezza > 3, e ordinate.

Obiect[] v = getStream()

.**distinct**() //toglie i duplicati

.**filter**(s->s.length() > 3) //elimina le stringhe con lunghezza <=3

.**sorted**() // ordina alfabeticamente

.**toArray();**

v contiene "bianco", "giallo", "grigio", "nero", "rosso", "verde"

Object[] **toArray** (): operazione finale che dà l'array contenente gli elementi dello stream.

Per ordinare in senso decrescente: .**sorted**(comparing(reverseOrder()))

Esempio

Esempio: fornire un array con le stringhe di lunghezza > 4, distinte, ordinate in senso inverso.

```
Object[] v = getStream()
.filter (s -> s.length() > 4) //tiene le parole con lunghezza > 4
.distinct() //elimina i duplicati
.sorted(comparing(s -> s, reverseOrder()))  //alfabetico inverso
.toArray();

System.out.println(Arrays.toString(v));
//[verde, rosso, grigio, giallo, bianco]
```

Esempi di count e max

Data la lista List<String> lista

contenente le parole seguenti: "bianco", "rosso", "blu", "verde", "grigio", "bianco", "nero", "verde", "giallo", "rosso"

si vuole contare le parole oppure ottenere la parola maggiore in senso alfabetico.

```
long l = lista.stream().count();  //10
```

```
Optional<String> max = lista.stream().max(naturalOrder());
System.out.println(max.get()); //verde
```

Optional<T> è un contenitore che può contenere un oggetto di tipo T.

Classe Optional<T> (in java.util)

Rappresenta un contenitore che può essere privo di contenuto. Se ha un valore, isPresent() dà true and get() dà il valore; orElse dà un valore di default se il contenitore è vuoto.

Metodi principali

static <T> Optional<T> empty(); genera un contenitore vuoto

static <T> Optional<T> of (T value); genera un c. con il valore dato

boolean **isPresent**()

T **get**() dà il valore contenuto; se manca lancia NoSuchElementException

T **orElse**(T other); se il contenitore è vuoto dà il valore indicato

Classi specifiche

OptionalInt con metodi specifici

static OptionalInt of(int value); int getAsInt(); int orElse(int other)

idem per **OptionalLong** e **OptionalDouble**

Uso di Optional<T>

Optional<String> max = lista.stream()**.max**(*naturalOrder()*);
String s = max.get(); //in generale può lanciare un'eccezione

oppure
String s = max.orElse(null); // dà null se il contenitore è vuoto

Operazioni intermedie di mapping

mapping da T a R

<R> Stream<R> **map** (Function<? super T,? extends R> mapper)

da T a Integer

IntStream **mapToInt** (ToIntFunction<? super T> mapper)

Note: ci sono anche mapToLong e mapToDouble

da T a Stream<R>

<R> Stream<R> **flatMap** (Function<? super T,
? extends Stream<? extends R>> mapper)

da T a IntStream

IntStream **flatMapToInt**(Function<? super T,? extends IntStream>
mapper)

Note: ci sono anche flatMapToLong e flatMapToDouble

Mapper

mapper 1 – 1 : da T o superclasse di T a V o sottoclasse di V

Function<? super T,? extends V>

mapper uno a molti, usato in flatMap
da T o superclasse di T a Stream di R (o sottoclasse di R)

Function<? super T,? extends Stream<? extends R>>

Esempio
da un ordine cliente ad uno stream di libri
ordine -> ordine.getlibri().stream()

getLibri dà la lista dei libri richiesti dall'ordine

Esempi

Calcolare la somma e la media delle lunghezze delle parole

```
int n = getStream()
.mapToInt(String::length) // da Stream<String> a IntStream
.sum(); //somma delle lunghezze delle parole:  51

OptionalDouble media = getStream()
.mapToInt(String::length)
.average(); // 5.1.
```

Collettori

Di solito il risultato dell'elaborazione di uno stream è dato mediante un collettore richiamato dall'operazione finale **collect**.

Il risultato può essere singolo: conteggio degli elementi dello stream, elemento maggiore o minore, somma o media di un attributo numerico degli elementi, concatenazione degli elementi (se sono stringhe).

Oppure può essere una collezione

o una partizione

o un raggruppamento: mappa (anche gerarchica).

La classe **Collectors** contiene numerosi metodi statici che generano dei collettori capaci di produrre i risultati indicati.

Classe ***Collectors***

La classe **Collectors** contiene numerosi metodi statici che generano dei collettori capaci di trattare le situazioni più frequenti.

Per utilizzarli facilmente occorre includere

```
import static java.util.stream.Collectors.*;
```

Metodi statici principali

counting, summingInt, averagingInt, maxBy, minBy, joining,

toCollection, toSet, toList, toMap,

groupingBy, partitioningBy,

mapping.

Esempi: counting

```
static Stream<String> getStream() {
return Stream.of("bianco", "rosso", "blu", "verde", "grigio",
"bianco", "nero", "verde", "giallo", "rosso"); }

//serve a fornire ogni volta lo stesso stream

conteggio degli elementi

long l = getStream().collect(counting());  //10

l = getStream().count(); //metodo principale equivalente
```

Esempi: summingInt, averagingInt, minBy

somma delle lunghezze degli elementi

int n = getStream().collect(**summingInt**(String::length)); //51

String::length è un mapper che dà il valore dell'attributo int da sommare

media delle lunghezze

double d = getStream().collect(**averagingInt**(String::length));//5.1

stringa minore

String min = getStream().collect(**minBy**(naturalOrder())).get(); //bianco

Senza get il risultato sarebbe di tipo Optional<String>

Esempi: joining

concatenazione delle stringhe

String s = getStream().collect(**joining**(", ")); //", " è il separatore

//bianco, rosso, blu, verde, grigio, bianco, nero, verde, giallo, rosso

s = getStream().collect(**joining**(", ", "[", "]")); [] inizio e fine del risultato

//[bianco, rosso, blu, verde, grigio, bianco, nero, verde, giallo, rosso]

Esempi: toList, toSet, toCollection

Lista come risultato

List<String> list = getStream().collect(**toList**());

Set come risultato (elementi distinti)

Set<String> set = getStream().collect(**toSet**()); // il toSet non ordina

[bianco, grigio, nero, rosso, giallo, blu, verde]

Per avere un treeSet occorre usare toCollection e passare un supplier

SortedSet<String> tset =
getStream().collect(**toCollection**(TreeSet::new));

[bianco, blu, giallo, grigio, nero, rosso, verde] //ordinato

Nota: **TreeSet::new** è un supplier in quanto dà un nuovo TreeSet con ordinamento alfabetico. Per avere un ord. decrescente

toCollection(() -> new TreeSet(reverseOrder()));

Collettore toMap

Colloca la sequenza di elementi in una mappa ed esiste in 3 versioni, con 2, 3 o 4 parametri.

I primi 2 parametri indicano come ricavare la chiave e il valore dagli elementi; dà errore se le chiavi sono duplicate.

```
Map<String, Integer> map =
        getStream().collect(toMap(String::toString, String::length));
```

dà l'eccezione java.lang.IllegalStateException: Duplicate key 6

perché le chiavi sono duplicate ad es. bianco.

Nota: la chiave è la stringa e il valore è la lunghezza.

Collettore toMap

Il terzo parametro permette di combinare due valori con la stessa chiave.

```
Map<String, Integer> map1 =getStream()
.collect(toMap(String::toString, String::length, (l1, l2) -> l1));
```

In questo caso non serve combinare le lunghezze; se ne tiene una.

Il quarto parametro permette di ordinare la mappa (mediante un supplier).

```
TreeMap<String, Integer> map2 = getStream()
.collect(toMap(String::toString, String::length, (l1, l2) -> l1,
        TreeMap::new));
```

La stampa delle due mappe:

```
{bianco=6, grigio=6, nero=4, giallo=6, rosso=5, verde=5, blu=3}
{bianco=6, blu=3, giallo=6, grigio=6, nero=4, rosso=5, verde=5}
```

Collettore partitioningBy

Suddivide gli elementi in due gruppi in base ad un *predicato* e dà come risultato una mappa booleana (con chiavi true e false).

Ci sono 2 varianti: nella prima ciascun gruppo è posto in una lista, nella seconda in un collettore fornito come secondo parametro.

```
Map<Boolean, List<String>> map3 = getStream()
.collect(partitioningBy(s->s.length() > 4));
{false=[blu, nero], true=[bianco, rosso, verde, grigio, bianco, verde, giallo, rosso]}

Map<Boolean, TreeSet<String>> map4 = getStream()
.collect(partitioningBy(s->s.length() > 4, toCollection(TreeSet::new)));
{false=[blu, nero], true=[bianco, giallo, grigio, rosso, verde]}
// senza duplicati
```

Collettore groupingBy

L'operazione finale spesso consiste nel fornire una mappa che raggruppa gli oggetti (o informazioni correlate) in base a chiavi.

Il metodo usato si chiama groupingBy e ha 3 forme in base al numero dei parametri (da 1 a 3).

1 parametro: è una funzione che ricava le chiavi dagli oggetti. A parità di chiave inserisce gli oggetti in una lista.

Raggruppare i colori in base alla lunghezza

Map<Integer, List<String>> map = getStream()

.collect(**groupingBy**(String::length));

{3=[blu], 4=[nero], 5=[rosso, verde, verde, rosso], 6=[bianco, grigio, bianco, giallo]}

Nota: per avere la lista ordinata basta inserire l'operazione sorted prima della collect.

Collettore groupingBy

2 parametri

Il secondo parametro è un collettore che indica come trattare gli oggetti associati alle chiavi.

Contare le occorrenze delle parole con una mappa (n. occorrenze per parola); il secondo collettore effettua un counting.

```
Map<String, Long> map = getStream()
.collect(groupingBy(s -> s, counting()));
```

{bianco=2, grigio=1, nero=1, giallo=1, rosso=2, verde=2, blu=1}

La funzione che dà la chiave è s -> s; la stringa stessa è la chiave

oppure si può usare String::toString (method reference)

Collettore groupingBy

3 parametri

Il secondo parametro di prima diventa il terzo parametro; il secondo è un supplier che dà il tipo di mappa desiderato, ad es. un treeMap con ordinamento naturale o esterno (con comparatore).

Come il precedente ma con le chiavi ordinate.

```
SortedMap<String, Long> map = getStream()
.collect(groupingBy(s -> s, TreeMap::new, counting()));
```

```
{bianco=2, blu=1, giallo=1, grigio=1, nero=1, rosso=2, verde=2}
```

Collettore groupingBy

Come il precedente ma con i colori ordinati in senso inverso

```
TreeMap<String, Long> m2 = getStream()
.collect(groupingBy(s -> s,
        ()-> new TreeMap<String, Long>(reverseOrder()),
        counting()));
System.out.println(m2);
```

```
{verde=2, rosso=2, nero=1, grigio=1, giallo=1, blu=1, bianco=2}
```

Nota: il secondo parametro è un supplier che fornisce un TreeMap con ordinamento in reverseOrder. In questo contesto è meglio scrivere esplicitamente i tipi del TreeMap anziché usare la forma accorciata new TreeMap<>.

Collettore groupingBy gerarchico

Raggruppare le parole per lunghezza decrescente e a parità di lunghezza dare il n. di ripetizioni delle parole ordinate.

```
SortedMap<Integer, SortedMap<String, Long>> m3 = getStream()
.collect(
        groupingBy(
                String::length,
                ()-> new TreeMap<Integer, SortedMap<String, Long>>
                        (reverseOrder()),
                groupingBy(String::toString, TreeMap::new, counting())
        )
);
System.out.println(m3);
```

{6={bianco=2, giallo=1, grigio=1}, 5={rosso=2, verde=2}, 4={nero=1}, 3={blu=1}}

Esempi con oggetti strutturati: lista di libri

Questi esempi servono per un riepilogo delle operazioni sugli stream e in particolare per illustrare le operazioni map e flatMap e il collettore mapping.

Gli esempi riguardano una collezione di libri e una collezione di ordini cliente.

Un libro ha gli attributi seguenti:

autore, titolo, editore (di tipo String), pagine (int).

es. new Libro ("falco", "rosso", "deltaplano", 100)

Un ordine cliente contiene il nome del cliente e la lista dei libri ordinati.

Classe Libro

```
public class Libro {
private String autore; private String titolo;
private String editore; private int pagine;

public Libro(String autore,String titolo,String editore,int pagine) {…}


public String toString() {return titolo;}
```

+ metodi getter e setter

Classe Libro

```
public static Libro[] libri = {
new Libro ("falco", "rosso", "deltaplano", 100),
new Libro ("falco", "arancio", "caravella", 120),
new Libro ("rondine", "blu", "caravella", 250),
new Libro ("rondine", "azzurro", "caravella", 250),
new Libro ("rondine", "indaco", "deltaplano", 80),
new Libro ("rondine", "giallo", "deltaplano", 100),
new Libro ("corvo", "nero", "monociclo", 300),
new Libro ("corvo", "grigio", "caravella", 240),
};
public static List<Libro> listaLibri = Arrays.asList(libri);
```

Per ottenere uno stream dalla lista dei libri basta scrivere
Libro.listaLibri.stream()

Esempi: map

```
//lista dei titoli con più di 100 pagine
List<String> titoli = Libro.listaLibri.stream()
.filter(libro -> libro.getPagine() > 100)
.map(Libro::getTitolo) //da stream di libri a stream di titoli
.collect(toList());
```

[arancio, blu, azzurro, nero, grigio]

L'operazione map fa corrispondere ad ogni oggetto dello stream un altro oggetto che si ottiene con una funzione di mapping (il parametro della map). Il mapping è 1-1.

Esempi: joining

```
//stringa con i titoli ordinati separati da virgola e spazio
//racchiusa tra graffe
String titoli = Libro.listaLibri.stream()
.map(Libro::getTitolo)
.sorted()
.collect(joining(", ", "{", "}"));
```

{arancio, azzurro, blu, giallo, grigio, indaco, nero, rosso}

Nota: parametri della joining

joining(CharSequence **delimiter**, CharSequence **prefix**, CharSequence **suffix**)

oppure joining(CharSequence **delimiter**)

Esempi: summingInt

uso del collettore summingInt

```
//numero totale di pagine dei libri
int totale = Libro.listaLibri.stream()
.collect(summingInt(Libro::getPagine));
```

1440

Esempi: groupingBy

```
//raggruppare libri per autore
Map<String, List<Libro>> mappa = Libro.listaLibri.stream()
.collect(groupingBy(Libro::getAutore));
```

```
{rondine=[blu, azzurro, indaco, giallo], falco=[rosso, arancio], corvo=[nero, grigio]}
```

Esempi: groupingBy

```
//libri raggruppati per autore e poi per n. di pagine mappa doppia
Map<String, Map<Integer,List<Libro>>> mappa = Libro.listaLibri.stream()
.collect(groupingBy(Libro::getAutore, groupingBy(Libro::getPagine)));
```

```
{rondine={80=[indaco], 100=[giallo], 250=[blu, azzurro]}, falco={100=[rosso],
120=[arancio]}, corvo={240=[grigio], 300=[nero]}}
```

```
//come sopra ma con tutti gli elementi ordinati
SortedMap<String, SortedMap<Integer,List<Libro>>> mappa =
Libro.listaLibri.stream()
.sorted(comparing(Libro::getTitolo)) //per avere le liste ordinate
.collect(groupingBy(Libro::getAutore, TreeMap::new,
groupingBy(Libro::getPagine, TreeMap::new, toList())));
```

```
{corvo={240=[grigio], 300=[nero]}, falco={100=[rosso], 120=[arancio]},
rondine={80=[indaco], 100=[giallo], 250=[azzurro, blu]}}
```

Esempi: summingInt, partitioningBy

```
//n. di pagine dei libri raggruppati per autore con autori ordinati
SortedMap<String, Integer> mappa = Libro.listaLibri.stream()
.collect(groupingBy(Libro::getAutore, TreeMap::new,
summingInt(Libro::getPagine)));
```

{corvo=540, falco=220, rondine=680}

```
//partizionare i libri in base al n. di pagine (<= 200 o > 200)
Map<Boolean, List<Libro>> mappa = Libro.listaLibri.stream()
.collect(partitioningBy(l -> l.getPagine() <= 200));
```

{false=[blu, azzurro, nero, grigio], true=[rosso, arancio, indaco, giallo]}

Esempi: maxBy

```
//il più spesso dei libri raggruppati per autore
Map<String, Optional<Libro>> mappa = Libro.getStream()
.collect(groupingBy(Libro::getAutore, TreeMap::new,
        maxBy(comparing(Libro::getPagine))));
```

```
{corvo=Optional[nero], falco=Optional[arancio], rondine=Optional[blu]}
```

Collettore mapping

Il collettore mapping raggruppa oggetti secondari, ad es. i titoli dei libri dato uno stream di libri.

```
List<String> l1 = Libro.listaLibri.stream()
        .collect(mapping(Libro::getTitolo, toList()));
```

[rosso, arancio, blu, azzurro, indaco, giallo, nero, grigio]

I libri sono gli oggetti primari. La lista però contiene non i libri ma i titoli (oggetti secondari). Il mapping ha due parametri: la funzione (mapper) che dà l'oggetto secondario da quello primario e il collettore che tratta gli oggetti secondari.

Esempi: mapping

```
//raggruppare per autore i titoli dei libri
Map<String, List<String>> mappa = Libro.listaLibri.stream()
.collect(groupingBy(Libro::getAutore,
mapping(Libro::getTitolo, toList())));  // da libri a titoli
```

```
{rondine=[blu, azzurro, indaco, giallo], falco=[rosso, arancio], corvo=[nero, grigio]}
```

```
//raggruppare per autore i titoli in un set ordinato
Map<String, TreeSet<String>> mappa = Libro.listaLibri.stream()
.collect(groupingBy(Libro::getAutore,
mapping(Libro::getTitolo, toCollection(TreeSet::new))));
```

```
{rondine=[azzurro, blu, giallo, indaco], falco=[arancio, rosso], corvo=[grigio, nero]}
```

Esempi: mapping

```
//mappa ordinata autori con il n. di pagine del libro più spesso

Map<String, Optional<Integer>> mappa = Libro.getStream()
.collect(groupingBy(Libro::getAutore, TreeMap::new,
        mapping(Libro::getPagine,maxBy(comparing(Integer::intValue)))
));
```

```
{corvo=Optional[300], falco=Optional[120], rondine=Optional[250]}}
```

Analisi degli ordini cliente con flatMap

Classe OrdineCliente

flatMap

Esempi

Stream sequenziali separati o congiunti

Classe OrdineCliente

Un OrdineCliente contiene il nome del cliente e la lista dei libri.

```
public class OrdineCliente {
private String cliente; private List<Libro> libri;

public OrdineCliente(String cliente, Libro... libri) {
        this.cliente = cliente; this.libri = Arrays.asList(libri);
}
public String getCliente() {return cliente;}
public List<Libro> getlibri() {return libri;}
```

Nota

Arrays.asList converte un array in una lista (con lunghezza fissa).

Classe OrdineCliente

```
public static OrdineCliente[] ordiniCliente = {
new OrdineCliente("quercia", Libro.libri[0], Libro.libri[1], Libro.libri[2],
Libro.libri[3], Libro.libri[4], Libro.libri[5], Libro.libri[6],Libro.libri[7]),

new OrdineCliente("olmo", Libro.libri[0], Libro.libri[1]),

new OrdineCliente("acero", Libro.libri[4], Libro.libri[5], Libro.libri[6])
};
public static Stream<OrdineCliente> getStream() {return
        Stream.of(ordiniCliente);}
}
```

Operazione flatMap

Trasforma uno stream di elementi X che hanno componenti Y

in uno stream di elementi Y.

Esempio: da uno stream di ordini cliente dove ogni ordine si riferisce ad uno o più libri si passa ad uno stream di libri.

.flatMap(ordine -> ordine.getLibri().stream())

Definizione di flatMap

<R> Stream<R>

flatMap(Function<? super T,? extends Stream<? extends R>> **mapper**)

Versioni numeriche

IntStream **flatMapToInt** (Function<? super T,? extends IntStream> mapper)

anche **flatMapToLong** e **flatMapToDouble**

Esempi: flatMap

```
//lista dei libri richiesti dagli ordini cliente
List<Libro> lista = OrdineCliente.getStream()
.flatMap(ordine -> ordine.getLibri().stream())
.collect(toList());
```

[rosso, arancio, blu, azzurro, indaco, giallo, nero, grigio, rosso, arancio, indaco, giallo, nero]

```
//fornire i numeri di copie per titolo
Map<String, Long> mappa = OrdineCliente.getStream()
.flatMap(ordine -> ordine.getLibri().stream())
.collect(groupingBy(Libro::getTitolo, counting()));
```

{azzurro=1, arancio=2, grigio=1, nero=2, indaco=2, giallo=2, rosso=2, blu=1}

Esempi: flatMap

//raggruppare per editore i numeri di copie per titolo

mappa gerarchica

```
Map<String, Map<String, Long>> mappa = OrdineCliente.getStream()
.flatMap(ordine -> ordine.getLibri().stream())
.collect(groupingBy(Libro::getEditore, groupingBy(Libro::getTitolo, counting())));
```

```
{caravella={azzurro=1, arancio=2, grigio=1, blu=1}, deltaplano={indaco=2, giallo=2, rosso=2}, monociclo={nero=2}}
```

Esempio di stream sequenziali separati

Raggruppare i titoli ordinati per numero di copie in ordine crescente.

Nota: il n. di copie va calcolato e quindi serve una prima mappa che associ il n. di copie ai titoli ordinati.

```
SortedMap<String, Long> m1 = OrdineCliente.getStream()
.flatMap(ordine -> ordine.getLibri().stream())
.collect(groupingBy(Libro::getTitolo, TreeMap::new, counting()));
```

Poi con lo stream di entry della prima mappa (la chiave è il titolo e il valore è il n. di copie) si raggruppano in una seconda mappa i titoli per n. di copie ordinati.

```
SortedMap<Long, List<String>> mappa = m1.entrySet().stream()
.collect(groupingBy(e -> e.getValue(), TreeMap::new,
        mapping(e -> e.getKey(), toList())));
```

{1=[azzurro, blu, grigio], 2=[arancio, giallo, indaco, nero, rosso]}

Si passa quindi da SortedMap<String, Long> a

SortedMap<Long, List<String>>

Esempio di stream sequenziali congiunti

```
//raggruppare i titoli ordinati per numero di copie in ordine crescente
SortedMap<Long, List<String>> mappa = OrdineCliente.getStream()
.flatMap(ordine -> ordine.getLibri().stream())
.collect(groupingBy(Libro::getTitolo, TreeMap::new, counting()))
.entrySet().stream()
.collect(groupingBy(e -> e.getValue(), TreeMap::new,
        mapping(e -> e.getKey(), toList())));
```

La prima mappa contiene il n. di copie per titoli ordinati:
SortedMap<String, Long>.
Il secondo stream è costituito dalle entry della prima mappa
Map.Entry<String, Long>.
La seconda mappa ha come chiave il n. di copie (in ordine crescente) e come valore la lista ordinata dei titoli (ord. dovuto al primo TreeMap).

```
{1=[azzurro, blu, grigio], 2=[arancio, giallo, indaco, nero, rosso]}
```

Generazione degli ordini editore dagli ordini cliente

Definizione di OrdineEditore

Elaborazione degli ordini cliente e produzione di una lista di ordini editore

Ordini editori

Un OrdineEditore contiene il nome dell'editore e una lista di linee d'ordine.

Una linea contiene il titolo del libro e il n. di copie richieste.

Il costruttore riceve il nome dell'editore; il metodo addLinea aggiunge una nuova linea all'ordine.

La classe Linea è interna.

classe OrdineEditore

```
public class OrdineEditore {
private String editore;
private ArrayList<Linea> linee = new ArrayList<Linea>();
private class Linea {
        String titolo; int n;
        Linea(String titolo, int n) {this.titolo = titolo; this.n = n;}
        public String toString() {return titolo + ":" +n;}
}
public OrdineEditore (String editore) {
        this.editore = editore;
}
public void addLinea(String titolo, int n) {
        linee.add(new Linea(titolo, n));
}
public String toString() {return editore + " " + linee;}
```

Ordini editori

Si costruiscano gli ordini editori relativi agli ordini clienti.

1. Si definisce la mappa doppia che raggruppa per editore i titoli e i n. di copie.

```
Map<String, Map<String, Long>> mo = OrdineCliente.getStream()
.flatMap(ordine -> ordine.getLibri().stream())
.collect(groupingBy(Libro::getEditore, groupingBy(Libro::getTitolo, counting())));
```

{caravella={azzurro=1, arancio=2, grigio=1, blu=1}, deltaplano={indaco=2, giallo=2, rosso=2}, monociclo={nero=2}}

Ordini editori

2. Si elabora l'entrySet della prima mappa generando un ordine editore data la key e aggiungendo le linee in base al value (mappa titolo-n. di copie)

```
List<OrdineEditore> list = m0.entrySet().stream()
.map(e -> {   //le entry sono mappate in ordini editore
        OrdineEditore o = new OrdineEditore(e.getKey());
        e.getValue().forEach((k,v)-> o.addLinea(k, v.intValue()));
        return o;
})
.collect(toList());
```

La key contiene l'editore e il value la mappa titolo-n. di copie.

[caravella [azzurro:1, arancio:2, grigio:1, blu:1], deltaplano [indaco:2, giallo:2, rosso:2], monociclo [nero:2]]

Nota: le due elaborazioni si possono accorpare.

Uso di stream per fornire informazioni tramite un'interfaccia

Dati i libri inseriti negli ordini cliente si vuole fornire una lista di informazioni contenenti il titolo e il n. di copie. Le informazioni si devono conformare all'interfaccia InfoI.

```
public interface InfoI {
        String getNome();
        long getValore();
        default String getInfo() {
                return getNome() +": " + getValore() + " ";
        }
        static void stampaListaInfoI(String nomeLista, List<InfoI> lista) {
                System.out.println(nomeLista);
                for (InfoI i:lista) System.out.print(i.getInfo());
                System.out.println();
        }
}
```

Classe Info

Le informazioni sono fornite tramite la classe Info che implementa InfoI.

```
class Info implements InfoI{
        private String nome;
        private long valore;
        public String getNome() {return nome;}
        public long getValore() {return valore;}
        public Info(String nome, long valore)
                {this.nome = nome; this.valore = valore;}
}
```

Risultato

```
List<InfoI> li = OrdineCliente.getStream()
.flatMap(ordine -> ordine.getLibri().stream())
.collect(groupingBy(Libro::getTitolo, counting())) //Map<String, Long>
.entrySet().stream()
.map(e -> new Info (e.getKey(), e.getValue()))
.collect(toList());

InfoI.stampaListaInfoI("lista", li);
```

lista

azzurro: 1 arancio: 2 grigio: 1 nero: 2 indaco: 2 giallo: 2 rosso: 2 blu: 1

Nota: l'operazione map trasforma

uno stream di Map.Entry<String, Long> in uno stream di Info

Riepilogo

Metodi, collettori

Analisi della groupingBy

Esempi

Metodi principali di Stream<T>

intermedi (l'output è dello stesso tipo)

distinct, filter, sorted

intermedi (l'output è di tipo diverso)

map, mapToInt, mapToLong, mapToDouble

flatMap, flatMapToInt, flatMapToLong, flatMapToDouble

finali

collect, count, forEach, max, min, toArray

Collettori principali

La classe **Collectors** contiene numerosi metodi statici che generano dei collettori capaci di trattare le situazioni più frequenti.

Per utilizzarli facilmente occorre includere

```
import static java.util.stream.Collectors.*;
```

counting, summingInt, averagingInt, maxBy, minBy, joining,

producono un oggetto

toCollection, toSet, toList, toMap,

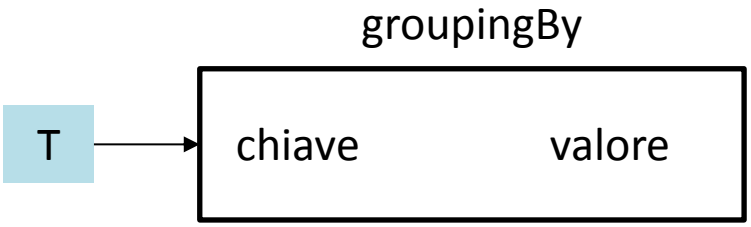
producono una collezione di T

groupingBy, partitioningBy, mapping.

Gli ultimi 3 contengono collettori.

Che cosa produce la groupingBy?

Produce una mappa la cui chiave è determinata dal primo parametro (un attributo di T). Ma i valori che struttura possono avere?



Un valore risulta dall'elaborazione di un gruppo di T (gli elementi con la stessa chiave) da parte del collettore della groupingBy.
Se manca il collettore il valore è una lista di T. Altrimenti può essere il n. di elementi del gruppo (counting), la somma o media dei valori di un attributo, l'elemento massimo o minimo del gruppo, una stringa (con joining se T è String); oppure una collezione.
Se il collettore è un groupingBy si ottiene una mappa gerarchica: ad es. gli abitanti di una nazione si possono suddividere per regione, provincia, città, se regione, provincia e città sono attributi degli abitanti.
Se il collettore è un mapping succede che si passa dal tipo T ad un altro tipo, T1, e il gruppo dei T1 è trattato dal collettore del mapping come illustrato in precedenza.
Il partitioning è una caso particolare di groupingBy con due chiavi: true e false.

Esempi

Libri raggruppati per autore e poi per numero di pagine

```
Map<String, Map<Integer,List<Libro>>> mappa =
Libro.listaLibri.stream()
.collect(groupingBy(Libro::getAutore, groupingBy(Libro::getPagine)));
```

n. di pagine dei libri raggruppati per autore con autori ordinati

```
SortedMap<String, Integer> mappa = Libro.listaLibri.stream()
.collect(groupingBy(Libro::getAutore, TreeMap::new,
summingInt(Libro::getPagine)));
```

il più spesso dei libri raggruppati per autore

```
Map<String, Optional<Libro>> mappa = Libro.getStream()
.collect(groupingBy(Libro::getAutore, TreeMap::new,
        maxBy(comparing(Libro::getPagine))));
```

Esempi

raggruppare per autore i titoli dei libri

```
Map<String, List<String>> mappa = Libro.listaLibri.stream()
.collect(groupingBy(Libro::getAutore,
mapping(Libro::getTitolo, toList())));  // da libri a titoli (stringhe)
```

Approfondimenti

Definizione dei collettori principali

Collectors

static <T> Collector<T,?,Long> **counting**()

static <T> Collector<T,?,Integer>
summingInt(ToIntFunction<? super T> mapper)

analogamente per summingLong, summingDouble

// il valore di T da sommare è ottenuto con un mapper a Integer

static <T> Collector<T,?,Double>

averagingInt(ToIntFunction<? super T> mapper)

analogamente per averagingLong, averagingDouble Double

Nota: il tipo del risultato è sottolineato.

Collectors

static Collector<CharSequence,?,String> **joining**()

//concatena le CharSequence dello stream

static Collector<CharSequence,?,String> joining(CharSequence delimiter) // con delimitatore

static Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)

// con delimitatore, prefisso e suffisso

Nota: **CharSequence** è un'interfaccia implementata da String, StringBuilder e altre classi.

static <T> Collector<T,?,Optional<T>> **maxBy**(Comparator<? super T> comparator)

static <T> Collector<T,?,Optional<T>> **minBy**(Comparator<? super T> comparator)

Collectors

da T a Collection<T> o estensioni (set o liste)

static <T,C extends Collection<T>> Collector<T,?,C>
toCollection(Supplier<C> collectionFactory)

da T a Set<T>

static <T> Collector<T,?,Set<T>> **toSet**()

da T a List<T>

static <T> Collector<T,?,List<T>> **toList**()

da T a Map<K,U> dove K e U si ottengono da T con due mapper

static <T,K,U> Collector<T,?,Map<K,U>>
toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)

//le chiavi non devono essere duplicate

Collectors

static <T,K,U> Collector<T,?,Map<K,U>>

toMap(Function<? super T,? extends K> *keyMapper*, Function<? super T,? extends U> *valueMapper*)

static <T,K,U> Collector<T,?,Map<K,U>>

toMap(Function<? super T,? extends K> *keyMapper*, Function<? super T,? extends U> *valueMapper*, BinaryOperator<U> *mergeFunction*)

static <T,K,U,M extends Map<K,U>> Collector<T,?,M>
toMap(Function<? super T,? extends K> *keyMapper*,Function<? super T,? extends U> *valueMapper*, BinaryOperator<U> *mergeFunction*, Supplier<M> *mapSupplier*)

Collectors

static <T> Map<Boolean,List<T> **partitioningBy**(Predicate<? super T> *predicate*)

static <T,D,A> Map<Boolean,D> **partitioningBy**(Predicate<? super T> *predicate*, Collector<? super T,A,D> *downstream*)

La mappa ha due chiavi (true e false).

Nel primo caso di partitioning, gli elementi T sono suddivisi in due liste, una per chiave. Nel secondo caso i due gruppi di T sono passati a due collettori.

Collectors

static <T,K> Map<K, List<T>> Collector<T,?,Map<K,List<T>>>

groupingBy(Function<? super T,? extends K> *classifier*)

static <T,K,A,D>Map>K,D> Collector<T,?,Map<K,D>>

groupingBy(Function<? super T,? extends K> *classifier*, Collector<? super T,A,D> *downstream*)

static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M>

groupingBy(Function<? super T,? extends K> *classifier*, Supplier<M> *mapFactory*, Collector<? super T,A,D> *downstream*)

Per classifier si intende un metodo di T che fornisce un oggetto K; gli oggetti T sono raggruppati in base agli oggetti K.

Nel primo caso un gruppo di oggetti con lo stesso K è posto in una lista.

Nel secondo caso è passato ad un collettore e quindi si possono avere dei sottoraggruppamenti.

Nel terzo caso la mapFactory fornisce il tipo di mappa con l'ordinamento desiderato.

Definizione del collettore mapping

static <T,U,A,R> Collector<T,?,R> mapping (

Function<? super T,? extends U> *mapper*,

Collector<? super U,A,R> *downstream*)

Un collettore mapping riceve oggetti di tipo T e mediante un mapper li converte in oggetti di tipo U e li passa ad un collettore.

Per mapper si intende una Function che da un oggetto T produce un oggetto U; spesso l'oggetto U è un attributo dell'oggetto T.

Al collettore sono passati gli oggetti U anziché i T.